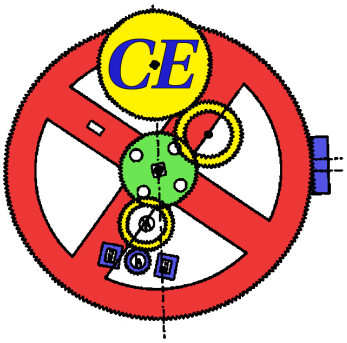


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2011

MSc THESIS

Free Viewpoint 3D TV Rendering Platform

Gokturk Cinserin

CE-MS-2011-22

Abstract

Over the past decade, products enabled by 3D technology have been increasingly adopted in the consumer market. The current challenge in the field is to explore the methods of free-viewpoint interpolation for 3D TVs. Free-viewpoint interpolation enhances the user experience significantly by allowing the viewer to select and alter the desired viewpoint of the scene interactively. This thesis covers the development of the free-viewpoint rendering platform for the European iGLANCE project. The proposed FTV platform is powered by a 7-issue slot VLIW architecture combining scalar and vector data paths. Our processor is based on an image signal processor (ISP) template from Silicon Hive. We vectorized the free-viewpoint algorithm developed in the context of the iGLANCE project, and mapped it onto this processor. The performance of our implementation is compared to an out-of-the-box implementation and previous vectorization work using the same architecture template. In order to address irregular memory accesses, identified as the bottleneck by the previous work, we used scatter-gather unit and a customized memory transfer scheme. This allowed us to apply several classical vectorization methods to fully utilize data level parallelism. In addition, instruction level parallelism is improved by applying further optimizations (loop transformations, data mapping, extending ISA). As a result of all above, a speed-up of a factor 6x is achieved over the selected baseline, which is equivalent to 78x over the out-of-the box code, and the ILP is improved by 17% as compared to the previous work. We set up a demonstration system to validate the results in a real-time environment by mapping our design to an FPGA running at 50 MHz frequency. The achieved frame rate is 6.75 fps in a 1280x720 resolution. This result indicates that when our design is mapped to silicon, running at about 10x the FPGA frequency and with extended processor resources, we will be able to achieve performance levels required in current-day consumer applications, which is Full HD resolution at 30 fps per eye.

TU Delft Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Free Viewpoint 3D TV Rendering Platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Gokturk Cinserin
born in Adana, Turkey

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Free Viewpoint 3D TV Rendering Platform

by Gokturk Cinserin

Abstract

Over the past decade, products enabled by 3D technology have been increasingly adopted in the consumer market. The current challenge in the field is to explore the methods of free-viewpoint interpolation for 3D TVs. Free-viewpoint interpolation enhances the user experience significantly by allowing the viewer to select and alter the desired viewpoint of the scene interactively. This thesis covers the development of an efficient free-viewpoint rendering platform for the European iGLANCE project. The proposed FTV platform is powered by a 7-issue slot VLIW architecture combining scalar and vector data paths. Our processor is based on an image signal processor (ISP) template from Silicon Hive. We vectorized the free-viewpoint algorithm developed in the context of the iGLANCE project, and mapped it onto this processor. The performance of our implementation is compared to an out-of-the-box implementation and previous vectorization work using the same architecture template. In order to address irregular memory accesses, identified as the bottleneck by the previous work, we used scatter-gather unit and a customized memory transfer scheme. This allowed us to apply several classical vectorization methods to fully utilize data level parallelism. In addition, instruction level parallelism is improved by applying further optimizations (loop transformations, data mapping, extending ISA). As a result of all above, a speed-up of a factor 6x is achieved over the selected baseline, which is equivalent to 78x over the out-of-the box code, and the ILP is improved by 17% as compared to the previous work. We set up a demonstration system to validate the results in a real-time environment by mapping our design to an FPGA running at 50 MHz frequency. The achieved frame rate is 6.75 fps in a 1280x720 resolution. This result indicates that when our design is mapped to silicon, running at about 10x the FPGA frequency and with extended processor resources, we will be able to achieve performance levels required in current-day consumer applications, which is Full HD resolution at 30 fps per eye.

Laboratory : Computer Engineering
Codenumber : CE-MS-2011-22

Committee Members :

Advisor:	Georgi Gaydadjiev, CE, TU Delft
Advisor:	Menno Lindwer, Silicon Hive
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

I dedicate this thesis to my grandmother, Meliha Cinserin, and my mother, Vildan Cinserin

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Menno Lindwer who guided and challenged me throughout the course of my thesis. This thesis could not have been a success without his contribution. I appreciate Georgi Gaydadjiev for his encouragement, patience, help and time. Without his guidance, I would not have a chance to be involved in this project.

I am grateful for the generous help provided by several people at Silicon Hive, Intel BV. I would specially like to mention Hendrik Boer, Mauro Cocco, Fiona Chua, and Alessandro Paschina for their support.

I am thankful to my parents, Vildan and Veysel Cinserin. Without their support I would not have had the strength to run this marathon. I am who I am because of them. Sizi cok seviyorum.

I thank my special cousins, and the rest of my great family. I am glad that I am a part of them.

I would also like to thank Aashini, Yunus, and Alper for standing by me through every happiness and dissappointment I have had this year. I also want to thank my friends from Delft for all the good times we had last year. Finally, I would like to thank all my friends in Turkey in memory of the days we had in the past, and for the days that we will have in the future.

Gokturk Cinserin
Delft, The Netherlands
September 1, 2011

1 Introduction

This chapter introduces 3D technology and the concept of free viewpoint interpolation. It also summarizes the main contribution of this work and provides an outline of the thesis.

This chapter is organized as follows. Section 1.1 starts with the background and motivation behind the Free Viewpoint 3D TV (FTV). Section 1.2 mentions the contributions of this work. Section 1.3 presents a brief outline of the thesis.

1.1 Background and Motivation

The emergence of 3D video, enabled by recent research and convergence of technologies from computer graphics, multimedia and related fields, enhanced the user experience by offering an advanced illusion of depth perception. The illusion of depth is achieved by the stereoscopy technique which is based on the idea of presenting two slightly offset 2D images to the left and right eye of the viewer. The human brain perceives 3D depth from these 2D images which are superimposed onto the same screen. The current trend in 3D viewer technology is to use passive (polarized, complementary color anaglyphs) or active (shutter) glasses in order to multiplex the view between the left and right eyes. Autostereoscopy is another method which offers to display 3D content without the use of special glasses by viewers.

Free Viewpoint TV (FTV) is an emerging field of study which expands the user experience even further, far beyond what is offered by traditional media. It allows the user to choose any viewpoint, thereby virtually navigating through a visual scene. As the total number of cameras recording the scene is limited, artificial views have to be created for all positions which do not have real camera data. The research on free viewpoint interpolation methods aims at rendering high quality artificial views within a feasible computation budget.

The iGLANCE project aims at researching methods of receiving and rendering free viewpoint video for both consumer 3D TV and healthcare domains. It also aims to actively contribute to the standardization process for the future 3D TV, and to facilitate, by a pertinent demonstration, the mass adoption and the commercial deployment of the future 3D TVs [20]. The project was launched in 2008 as a part of the European MEDEA+ program with a total budget of 18 million Euros shared between a French consortium and a Dutch consortium. Silicon Hive, as one of the project partners, is investigating how its processor technology can be extended for the field of 3D video and free viewpoint rendering.

This thesis focuses on the development of the free viewpoint rendering platform for the iGLANCE project. The key component which is responsible for viewpoint interpolation, in the proposed platform, is a Silicon Hive image signal processor (ISP). It is a VLIW processor equipped with a combination of scalar and vector issue slots. It enables

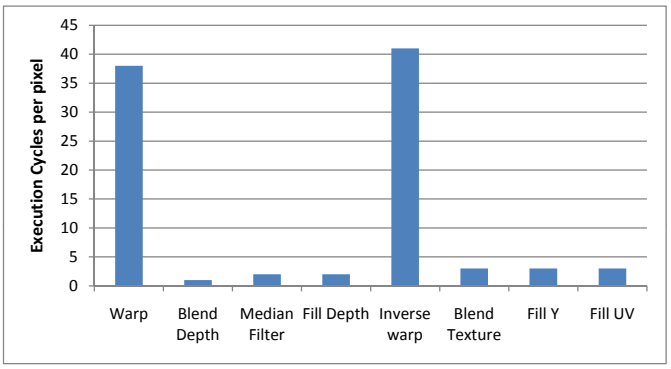


Figure 1.1: Performance of the previous vectorized implementation for each algorithm step

high levels of data and instruction level parallelism to achieve high real-time processing requirements. The free viewpoint algorithm, developed by Eindhoven University of Technology, Video Coding & Architectures research group, is vectorized and mapped onto this processor.

The previous implementation which was also based on Silicon Hive's ISP template was proposed in [6]. The bottleneck in this work is at the warping and inverse warping stages as shown in Figure 1.1. Irregular memory access patterns required by the pixel based load/stores prevents efficient vectorization of these stages hence causing a significant performance limitation. Therefore, the first focus is on the improvement of the warping stages, on which most of the processor cycles are spent, in order to overcome these limitations.

1.2 Contribution of this Thesis

- The main contribution of this work is the technique to handle irregular (random) memory accesses to allow efficient exploitation of data parallelism. The proposed technique is based on the use of scatter-gather units in combination with medium-size vector-addressable memories which are used for caching the frame data. Adopting these units along with the suggested method for caching and transferring the data between memories allow overcoming the issues related to vectorization.

- In order to benefit from the instruction level parallelism (ILP) in our VLIW architecture, several loop transformation techniques and an efficient data mapping are employed. Furthermore, an analysis is performed to extend the processor's instruction set with a number of custom instructions which are frequently used in the algorithm.

- Finally, a demonstration system, on an FPGA-based platform, is built in order to show the validity of the results of the implementation in a real-time environment as also aimed by the iGLANCE project.

1.3 Outline of the Thesis

Chapter 2 gives a more-detailed view of the iGLANCE project and background information about the Silicon Hive processor, system solutions and development flow. It also describes the previous work related to free-viewpoint rendering platforms before diving into the details of this work. This gets the reader acquainted with the motivations of the project.

The steps of the algorithm for performing free-view interpolation (FTV algorithm) are described in Chapter 3.

Chapter 4 is dedicated to explain the Silicon Hive ISP2400 processor based architecture on which the FTV algorithm is mapped. This chapter gives an overview about the individual components of the rendering platform; it also presents the synchronization method for these system components, and system-level performance estimations.

Chapter 5 explains the mapping of the FTV algorithm on the VLIW processor including the SIMD datapath. The challenges related to vectorizing the algorithm, and techniques to solve these challenges are explained in this chapter. Particularly, it addresses the key contribution of this thesis, which is the technique to improve the performance in the presence of irregular memory accesses. It also explains the optimizations which were applied to improve the performance of the algorithm further.

The results of the implementation are presented in Chapter 6, along with an analysis on the results and a comparison with previous implementations.

Chapter 7 concludes the thesis by summarizing the major contributions and the achieved results; and it suggests a number of points for further improvements.

Background Information and Related Work

2

This chapter starts by presenting a more detailed view of the iGLANCE project. The main goals of the project together with the iGLANCE architecture are explained. The target application domains of the project are given. The next section introduces Silicon Hive and their ASIP solutions which offer a competitive performance in the embedded domain. Silicon Hive processors, system solutions, and development flow is explained briefly in this section. Finally, we present the related research regarding free-viewpoint rendering platforms. We justify the advantages of our solution by comparing it to various other alternatives.

2.1 The iGLANCE Project

2.1.1 Overview

The goal of the iGLANCE project is to research and develop methods of receiving and rendering free-viewpoint in 3D-TVs [35]. It aims at defining an end-to-end 3D solution, and actively contributing to the standardization process of 3D TV. The project was launched in 2008 as a part of the European MEDEA+ program. MEDEA+ was initiated by industrial partners for co-operative R&D in microelectronics to improve Europe's competitiveness in this field. The iGLANCE project involves several partners from the industry as well as the Eindhoven University of Technology.

The project focuses on receiving and decoding multi-view streams on a decoding platform and free viewpoint interpolation on a rendering platform. The iGLANCE chain commences with the reception of the transmitted stream (see Figure 2.1). Based on the user input, the system ensures that only the parts which are necessary for view interpolation are transmitted. The captured stream is then decoded by H.264 decoder. After that, the iGLANCE system performs interpolation of the view chosen by the user. In the project, it is intended to propose the algorithms to realize the free-viewpoint selection, and to design and implement the hardware that is capable of decoding and free-viewpoint rendering of the video streams with real-time constraints.

2.1.2 The iGLANCE Architectural View

In order to validate the feasibility of the previously mentioned objectives, a demonstration system is implemented. The architectural components of the demonstrator are illustrated in Figure 2.2. The system is composed of a number of hardware and software components whose implementations are deployed on two separate boards. The front-end board is responsible for H.264 decoding of transmitted video streams. This board is provided by ST Microelectronics, an iGLANCE partner. The back-end board processes the decoded video stream in order to interpolate the view for the user-defined position.

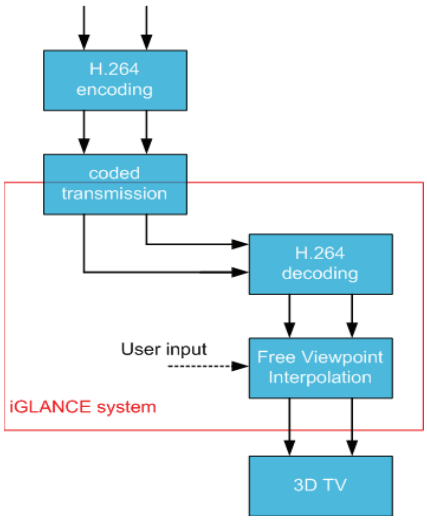


Figure 2.1: The iGLANCE transmission chain

Silicon Hive's Gladiator board is used for this purpose. A detailed explanation about the back-end architecture is given in Chapter 4.

The processing of data, i.e. decoding and interpolation, is realized by the hardware components of the system. In the demo system, the input is an encoded transport stream (.ts) from a USB stick. In the general case, the front end board receives the stream from a digital channel. After receiving the encoded video, the front end board transmits the decoded stream through an HDMI connection. It is captured by the HDMI interface of the back-end board, and then the final output is sent to the LCD display through the second HDMI connection.

Stacked above the hardware layer is a middleware which is a software-only component whose purpose is to combine the different system parts of the iGLANCE architecture such that a complete functional system exists. The middleware controls the iGLANCE video processing hardware by transferring the user commands such as video play, pause, stop, etc., to it. It also uploads the subset of parameters required for video processing, which are selected based on the user input. The communication of the middleware to the system hardware is provided using the application programmer's interface (API) of these hardware modules. The main part of the middleware is deployed on the front-end board, and it uses a proxy pattern in order to make remote procedure calls (RPC) to the C-functions present in the back-end middleware. This communication is done through an ethernet connection.

2.1.3 Scenarios and Requirements

The target application fields in the iGLANCE project are the consumer and the health-care domains. The consumer scenario FTV_Bypass aims at providing the 3D streams in full-HD (1920x1080) resolution at 30 fps per eye. The format of the stream can be sequential or side-by-side as demonstrated in Figure 2.3a and b respectively. The free

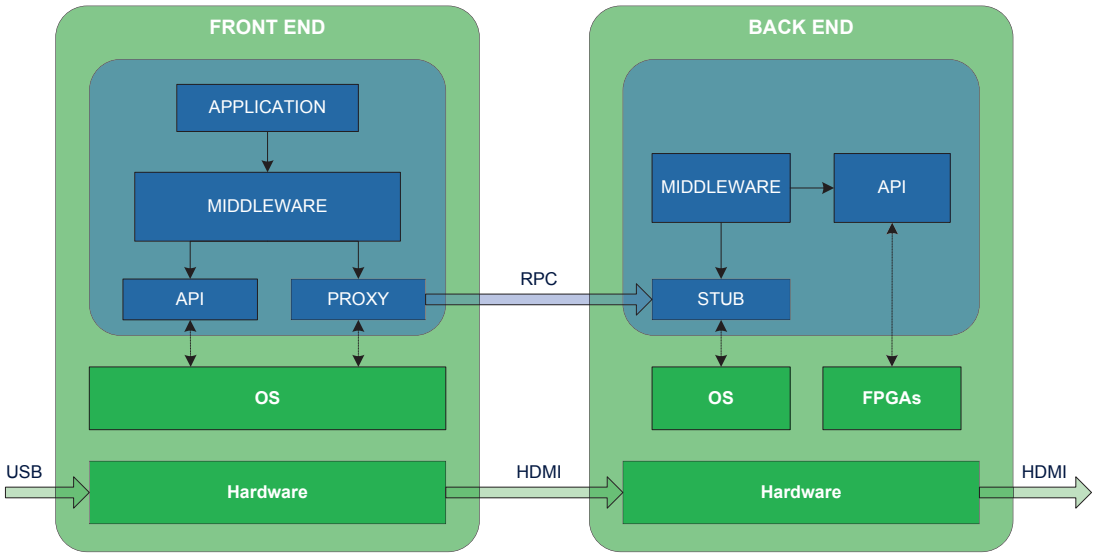


Figure 2.2: The iGLANCE architectural view

viewpoint interpolation on the back-end board is bypassed in this scenario. This is to meet the short-term requirements of the 3D home market.

The other consumer scenario FTV has been defined to realize a free-viewpoint selection feature based on the user input, using the views from the cameras recording the scene from left and right angles. The input is required to be a side-by-side texture and depth information from both cameras. The FTV algorithm (explained in Chapter 3) creates an artificial interpolated 2-D view between the two camera positions. By running the FTV algorithm twice, it is possible to obtain stereo views for the desired viewpoint. The interpolated stereo views, packed side-by-side, are sent to a 3D display. The requirement for this scenario is also a full-HD resolution at 30 fps per eye. The left and the right views generated by the FTV algorithm are of a resolution 960x1080, at 30 fps. The 3D panel is responsible for up-scaling each view to 1920x1080 and displaying them sequentially.

The healthcare scenario demonstrates the usefulness of the system in biomedical applications. In the medical domain, 3D data is generated by several imaging modalities. After processing, displaying this data on an auto-stereoscopic screen with a free-viewpoint option can help navigating instruments optimally during interventions [35]. In this scenario, the multiple interpolations for nine viewpoints, each with an SD resolution (640x360), packed into a full-HD frame has been proposed. This thesis focuses on the FTV consumer scenario.

2.1.4 Reference Video Sequence

Research in the field of multi-view and free-viewpoint rendering requires multi-view test sequences to be used as a reference by researchers. A widely-used video sequence is "ballet scene" generated and distributed by Microsoft Research Interactive Visual Group

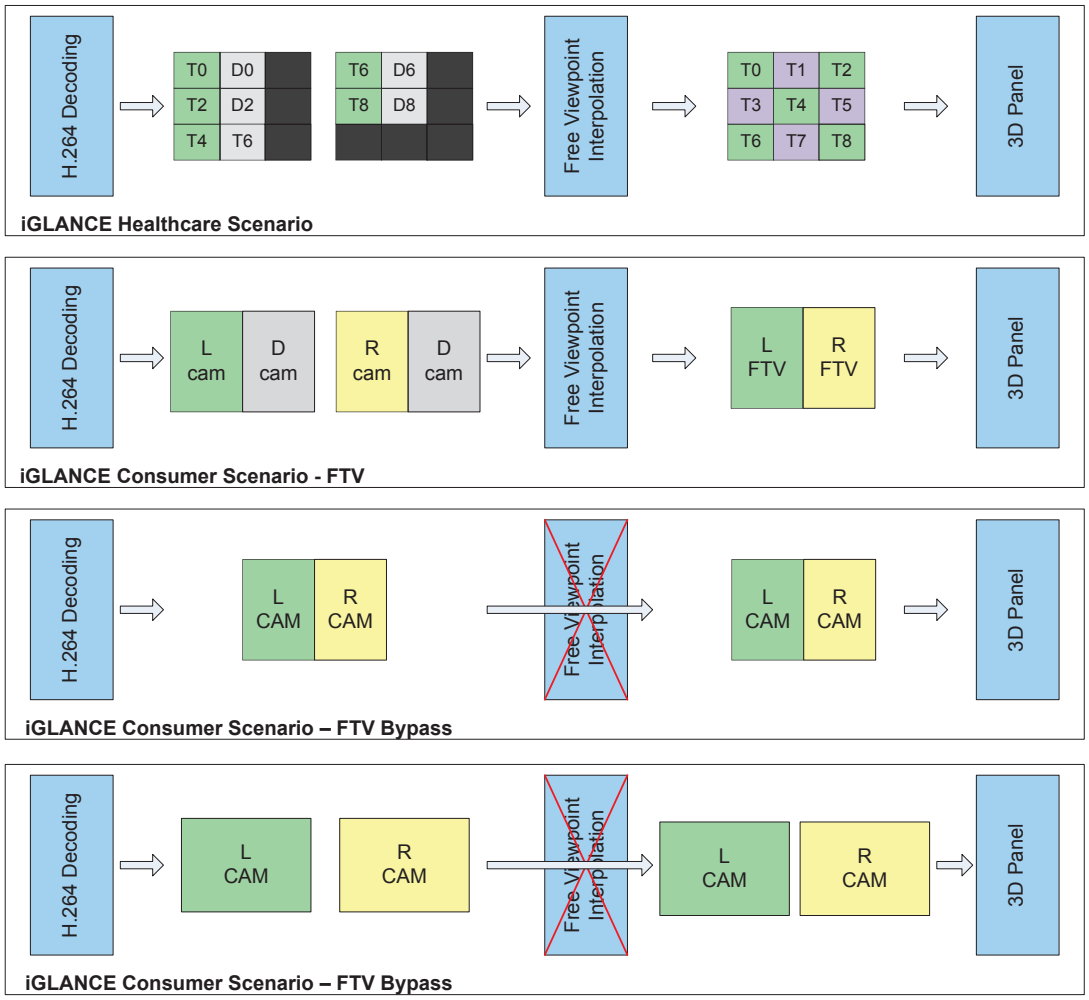


Figure 2.3: The iGLANCE scenarios

[36]. It is used in the iGLANCE project in order to test the algorithm.

The ballet scene was recorded using a set of synchronized cameras. There were eight cameras placed along an arc, covering about 30 degrees from one end to the other, as depicted in Figure 2.4 . This corresponds to an angle difference of 4 degrees between two consecutive cameras. The quality of the interpolated views generated by free-viewpoint algorithms depends on the distance between two camera views.

The ballet sequence contains 100 frames with a resolution of 1024x768 at 15 fps. The first frame of the sequence for the 4^{th} camera position and its corresponding depth map is shown in Figure 2.5.

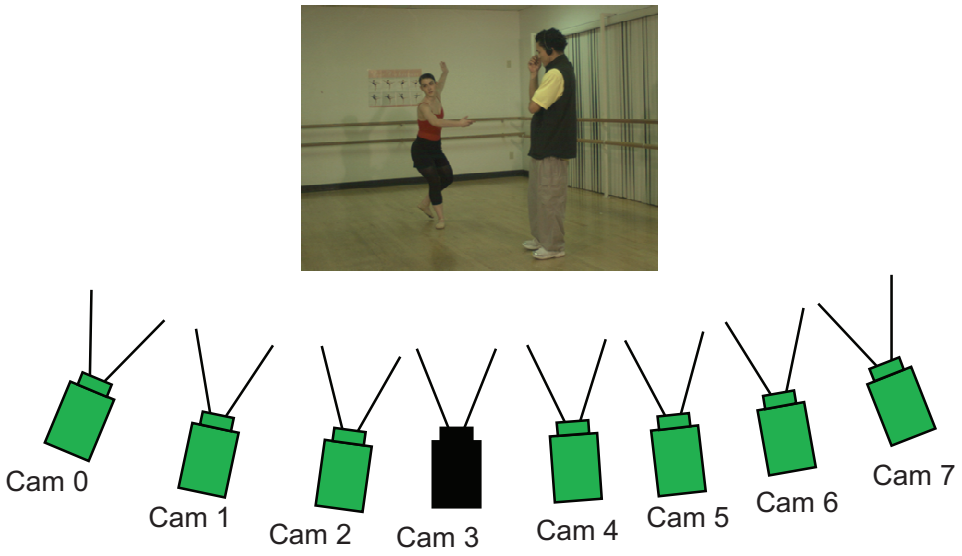


Figure 2.4: The camera set-up for recording the ballet scene

Figure 2.5: An example frame from the ballet scene

2.2 Silicon Hive Technology and Platforms

In the rapidly evolving embedded computing domain, general-purpose processors very often fail to satisfy the demanding power, area and performance requirements. This promoted the use of specialized embedded computing systems customized for specific set of applications. There has been a strong interest for Application Specific Integrated Circuits (ASIC) which is a dedicated hardware intended for a particular application. This specialization of an ASIC helps providing a very efficient solution to a particular problem in terms of performance and cost. But this advantage comes by sacrificing the variability of the device. The hardware must be redesigned and manufactured in case an update is needed for the application, which increases the overall costs and the time-to-market for the product. This, together with some other factors [22], gives rise to a shift

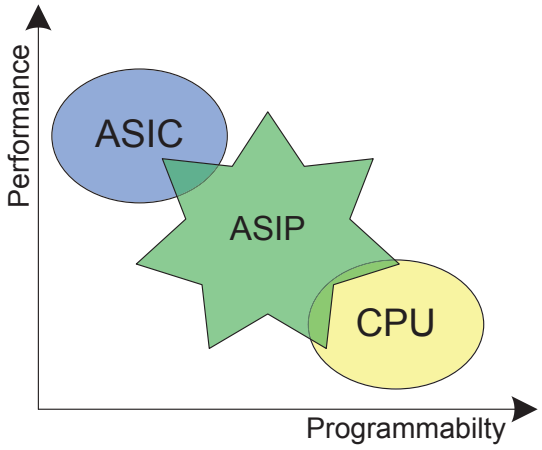


Figure 2.6: The spectrum of performance vs. programmability

from ASIC design towards the use of programmable platforms.

Application Specific Instruction-Set Processors (ASIP) represent application specific, yet programmable solutions. They represent the mid-point in the spectrum between very efficient, less flexible ASICs and less efficient, very flexible general-purpose processors (see Figure 2.6) [3]. The programmability of an ASIP comes from its instruction-set architecture (ISA) while the competitive performance is provided by customized instructions and exploiting high levels of parallelism.

Silicon Hive licences ASIP processors and systems offering very efficient and highly-programmable solutions. Besides that, Silicon Hive provides its customers with an advanced development environment which further reduces the time-to-market of products. Furthermore, the development environment inside Silicon Hive offers the possibility to customize the processors and systems easily in order to satisfy the requirements of various applications. In the following sections, brief information about the Silicon Hive processors, systems, and development environment is given.

2.2.1 Processors

The Silicon Hive processors are based on a load-store architecture, and belong to the very long instruction word (VLIW) processors family. These processors are capable of issuing and executing multiple operations simultaneously thereby achieving high levels of instruction level parallelism (ILP). Every operation in the instruction is fetched to the corresponding issue slot. An issue slot consists of one or more functional units each of which can perform a group of different operations. The maximum possible ILP, that the processor can achieve, can be increased by increasing the number of issue slots, by executing more operations in a single cycle.

Unlike many conventional VLIW architectures, the Silicon Hive processors have an option to be equipped with single instruction multiple data (SIMD) issue slots. Therefore, they also provide the advantages of data level parallelism besides instruction level parallelism. With the use of N-way vector issue slots, an operation is performed on N

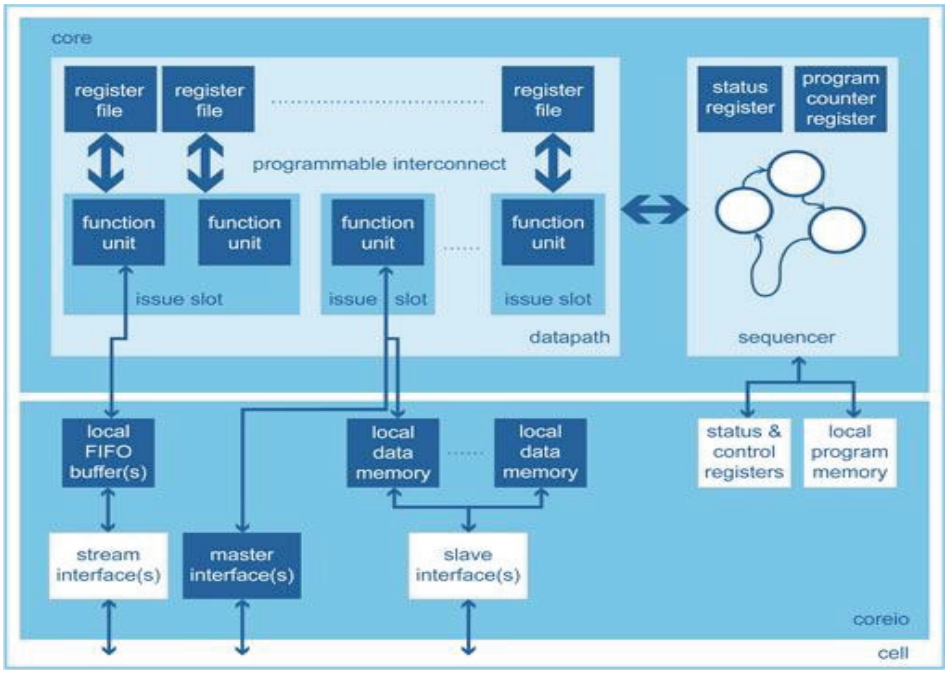


Figure 2.7: An example of a Silicon Hive VLIW processor

vector elements simultaneously. The functional units placed in the vector issue slots are capable of executing both inter and intra-vector operations. An example Silicon Hive processor is illustrated in Figure 2.7. The processor consists of issue slots, register files, a programmable interconnect, scalar and vector memories, core i/o interfaces, and a control logic. As opposed to most other processor designs, the Silicon Hive data path has multiple register files, usually at least one register file per issue slot to achieve low fan-in and fan-out [19]. The compiler handles the assignment of variables to various register files.

Silicon Hive does not focus on one complete solution for all domains, but rather it allows the processor designers to tune their core according to the target application, by exploring the design space and finding the best choice in terms of area, performance and power. Processors can be configured by varying any of aspects such as the number of issue slots, the types of functional units, the number and the size of register files, and the interconnect scheme between these components. This is made possible by providing a highly abstract language called TIM (The Incredible Machine) to configure the hardware elements of the processor. A few lines of TIM code can invoke pre-built hardware blocks written in hundreds of lines of VHDL code.

2.2.2 Systems

In the Silicon Hive development environment, every processor must be instantiated at system level in order to fulfill and test its functionality. A system is considered as a number of Silicon Hive processors, a host processor, a control bus, FIFO adapters,

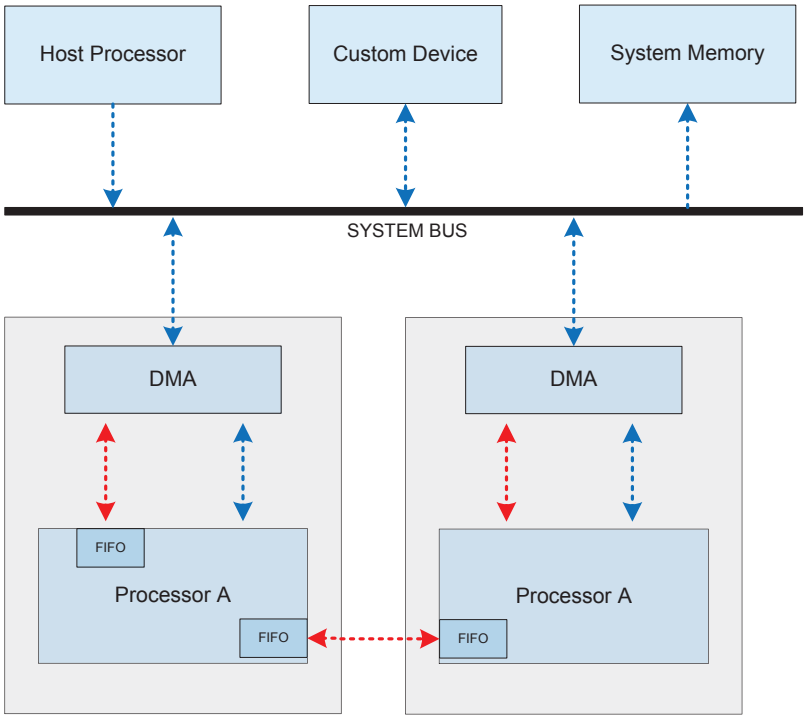


Figure 2.8: An example of a Silicon Hive multi-core system

external memories, and/or custom devices. Figure 2.8 demonstrates an abstract example of a multi-processor system including these components. A host processor, typically an ATOM or similar processor, is responsible for controlling the system. It initializes the other system components, uploads programs to the Hive cores, starts the execution of the program in these cores, and uploads required parameters to the processors' local memory or the external memory. The FIFOs are typically included for direct communication between processors and/or other hardware blocks. The data transfer between processors is performed through the shared bus. Each Hive processor in the figure has its own DMA to transfer data between the core and the external memory or the other cores.

In order to describe the Silicon Hive systems, a high-level language called HSD is used. A designer can use this language to configure the components, connections, and properties of a system in a hierarchical way. A system consists of a set of clusters which are composed of standard or custom devices. The system description is exploited to generate a system simulator containing all system-specific information, such as bus address mappings and connectivity. Using the same description, pre-built hardware blocks are invoked to map them on an FPGA or silicon.

2.2.3 The HiveCC Compiler

Unlike superscalar processors which use a hardware scheduler, the performance of the VLIW architectures highly relies on an effective instruction-scheduling compiler. The instruction scheduler packs all the operations that can be executed in parallel into a single,

very-large instruction word. Therefore, it determines to what extend the resources of a VLIW processor are utilized. VLIW processors can be built using extensive resources, but if a compiler is unable to schedule the instructions sufficiently parallel, then the processor resources are wasted.

Silicon Hive's HiveCC compiler is responsible for scheduling instructions and allocating the resources efficiently for the VLIW cores. It comes with two types of instruction schedulers, manifold and hivesched, both trying to achieve a schedule in as few instructions (processor cycles) as possible. The former tries to solve the scheduling problem, which is intractable, optimally. The latter is based on greedy algorithms which give a solution in shorter time but do not guarantee an optimal solution.

2.2.4 Silicon Hive Simulation Environment

Since the HiveCC compiler handles instruction scheduling and resource allocation, the performance of algorithms on a given architecture can be estimated. Silicon Hive's development environment offers different levels of simulation and verification for the target application. The choice between these levels depends on the purpose of simulation. They provide a trade-off between accuracy and simulation time. Table 2.1 illustrates four different simulation methods.

The first and the most abstract, c-run is the fastest method, which simulates only the functionality of the application. It consists of native compilation (using gcc compiler) of both a host program and functions that are to be accelerated by a Silicon Hive processor on a workstation. In an unsched run, the code is compiled with the HiveCC compiler, instruction selection is performed; however, the code is not scheduled. Therefore, information on cycle count and resource allocation is not present. The next level is the sched run where all resource conflicts are resolved, and a cycle accurate simulation regarding core operations takes place. This simulation uses the Silicon Hive compiled simulator model. In this flow, the code is compiled onto the assembly code and scheduled with the HiveCC, which is then compiled onto a C-function that cycle-accurately simulates the behavior of the intermediate code. This C-function is then linked with the host code to obtain a simulator for the application.The most accurate simulation method is system_vhdl which is performed at the signal level. Along with cycle accuracy at the core level, the cycle count in the core I/O level is also taken into account.

In a sched-run, the compiler generates an output html file containing the scheduling information with a visual interface. By analyzing this file, it is possible to observe the static usage of processor resources for each function in the code. This allows detecting the bottlenecks in the implementation, and boosting the performance by solving them. An example is given in the appendix.

2.3 Related Work

In the last decade, there has been extensive research on the implementation of a real-time rendering platform which is capable of doing free viewpoint 3D interpolation. Some research is focused on the implementation on a dedicated hardware (ASIC). In a relatively

Table 2.1: Different abstraction levels for simulation in Silicon Hive

Simulation Type	**Execution Method**	**Execution Time**	**Accuracy**
C-run	Native	Native	Functionality only
Unsched	Native	$>$ 3 Mops/s	Bit-accurate
Sched	Native	3 Mops/s	Cycle-accurate
System_vhdl	RTL	100 ops/s	Signal-accurate

old study [14], such a hardware accelerator capable of processing 25 SDTV (720x576) frames per second (fps) is proposed. In a more recent paper [32], an ASIC which can process up to 216 fps with a QFHD (4096x2160) resolution is presented. In [13], a multi-view rendering architecture is implemented for auto-stereoscopic displays using an Altera Cyclone III FPGA. Another widely-adopted solution is using high-end GPUs. [1] and [31] uses GPUs to achieve rendering at Full-HD (1920x1080) resolution with a frame rate of 30fps and 24 fps, respectively.

Among the proposed platforms, the dedicated hardware solutions offer the best performance to handle the extensive computation requirements. They also have the advantage of costing relatively less chip area along with superior power-efficiency. However, the hardcoded platforms imply a higher risk for future product improvements due to longer development cycle and high mask costs. This is a considerable drawback considering continuous improvements in the quality and the performance of the current rendering algorithms and the absence of a stable free viewpoint 3D-TV standard.

The FPGA-based implementations are mostly used for research purposes. The reconfigurability of such platforms presents a more suitable basis for design modifications. However, a flat learning curve in SW/HW mapping, insufficient processing power and a relatively high cost per item prevents an FPGA to be preferred as a mass production method for free viewpoint rendering devices [7].

The use of GPUs, on the other hand, promises high performance solutions while it also provides high flexibility for the modifications and optimizations on the implementation of the rendering algorithms. However, high power consumption figures and the high costs of high-end GPUs might become a bottleneck for adopting GPUs as a set-top-box for the free viewpoint rendering.

This thesis proposes an architecture based on Silicon Hive's ASIP solutions that offers a good combination of all the above approaches. Firstly, it engages high design flexibility by shifting the development cycle to software which is highly desired in this rapidly developing domain. It also presents a convenient way for prototyping purposes using existing Silicon Hive demonstration board with Xilinx Virtex-5 FPGAs. In this process, the processor and the rendering system can be tuned to maximize the performance. When the final design is mapped on silicon, the performance is competitive to that of an ASIC or a GPU while the power consumption is significantly less than GPU-based solutions, and its flexibility makes it more attractive than an ASIC.

3 FTV Algorithm

In this chapter, we first introduce some terminology related to the free-viewpoint algorithms, such as depth image based rendering, single-view vs. multi-view rendering. Then, we focus on getting the reader acquainted with the free-viewpoint algorithm that is used in this project. After giving an overview about the algorithm, we explain every step demonstrating result images from the sample ballet sequence

3.1 Overview

Generating different viewpoints to virtually walk through a scene, called as multi-view video, has become a hot topic in the field of advanced video processing [16]. As the total number of cameras recording the scene is limited, artificial views have to be created for all positions which do not have a real camera data. Free-viewpoint (FV) algorithms aim at rendering such artificial views. The latest trend in the FV algorithms involves the usage of the geometry of the scene in order to improve the rendering quality [29]. A well-known rendering technique called as the Depth Image Based Rendering (DIBR) uses a depth map of the scene as the geometry information. The purpose of the DIBR is to warp the single/multiple original image(s) into a synthetic view using the depth information. Research has been done on viewpoint interpolation from a single reference view and its corresponding depth map [26]. However, this method suffers from insufficient information and disocclusions in the resulting views. Some parts of the scene can be hidden in the reference view that is used, as shown in Figure 3.1. In fact, this is the fundamental challenge in viewpoint interpolation.

The other approach is to exploit two reference images and their depth maps in order

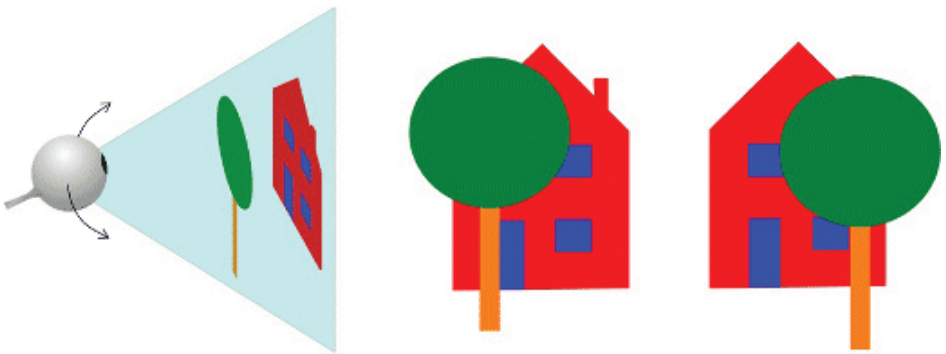


Figure 3.1: Disocclusions in the background depending on the viewpoint [34]

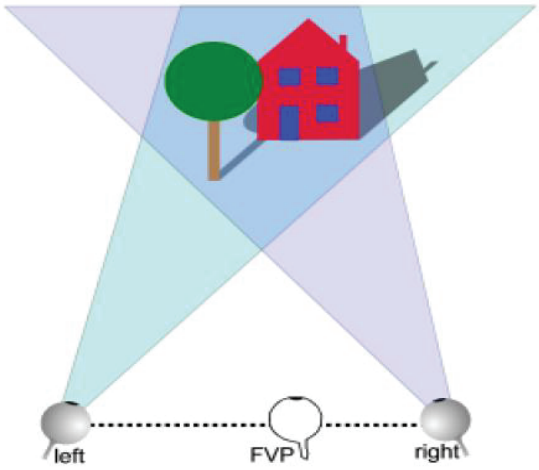


Figure 3.2: Interpolating a new view anywhere between two camera views [34]

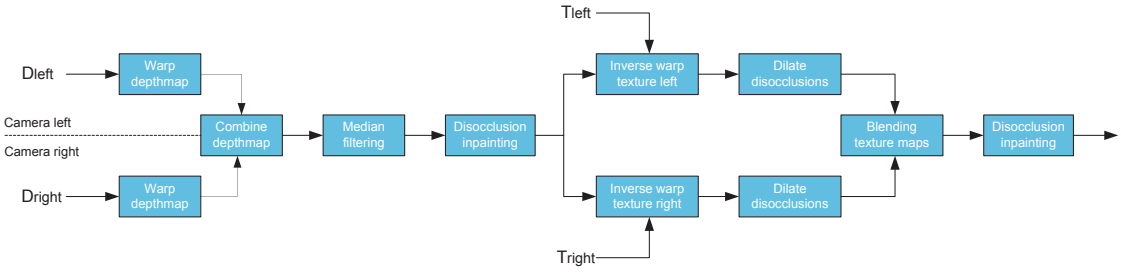


Figure 3.3: An overview of the FTV algorithm

to create a more accurate artificial view in between these two views (Figure 3.2). This method is called multi-view rendering. It has the advantage of being able to compensate for the information that is absent in the first view, from the other view. Therefore, it promises a better accuracy while the required number of computations is typically more than interpolation using single reference.

The algorithm used in this thesis is based on the multi-view rendering technique, and is developed by TU/e. Figure 3.3 illustrates the algorithm pipeline. First, the depth map for the desired viewpoint is generated through forward warping. Imperfections in this depth map are corrected by post-filtering and disocclusion inpainting. Then, the algorithm projects the input texture views into the virtual plane using the interpolated depth information for the user-defined viewpoint.

Note that all sample results shown in this chapter are obtained from the ballet scene using camera 2 and 4 as the left and the right reference cameras, as shown in Figure 2.1.4 . The output view is created for the viewpoint position where camera 3 is placed.

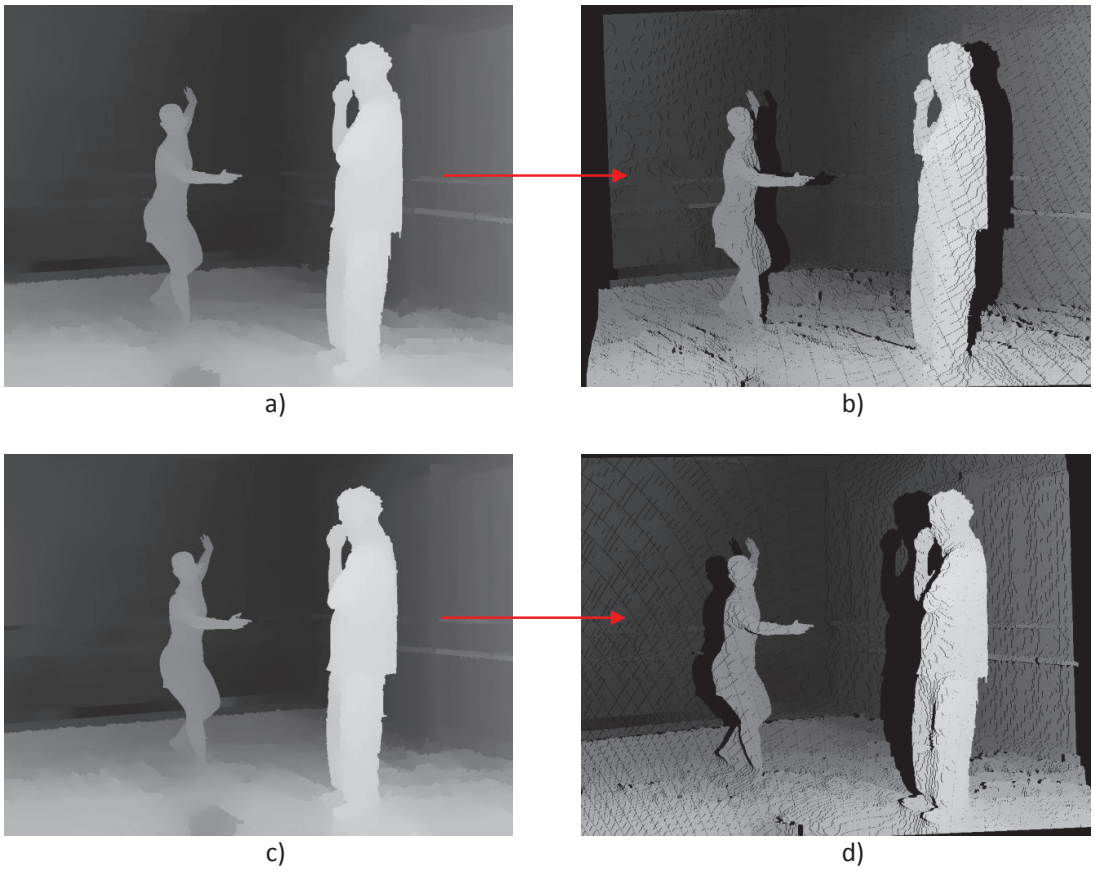


Figure 3.4: Warping depth maps : a) Depth from the original left camera, b) Depth after warping the left camera, c) Depth from the original right camera, d) Depth after warping the right camera

3.2 Stages of the FTV Algorithm

3.2.1 Warping Depth Maps

In this step, the depth maps at the two nearest original camera positions are warped to obtain the corresponding depth map at the interpolated position which is typically chosen by the user. This is achieved by first projecting the plane at the original position to 3D space, and then back-projecting it to the virtual image plane. Figure 3.4 shows the resulting images of rendering for both the left and the right depth maps. During the warping stage, a number of pixels can be projected to the same pixel position in the interpolated image. In such cases, the pixel with a lower depth value, i.e. nearer to the camera, has to be adopted [24].

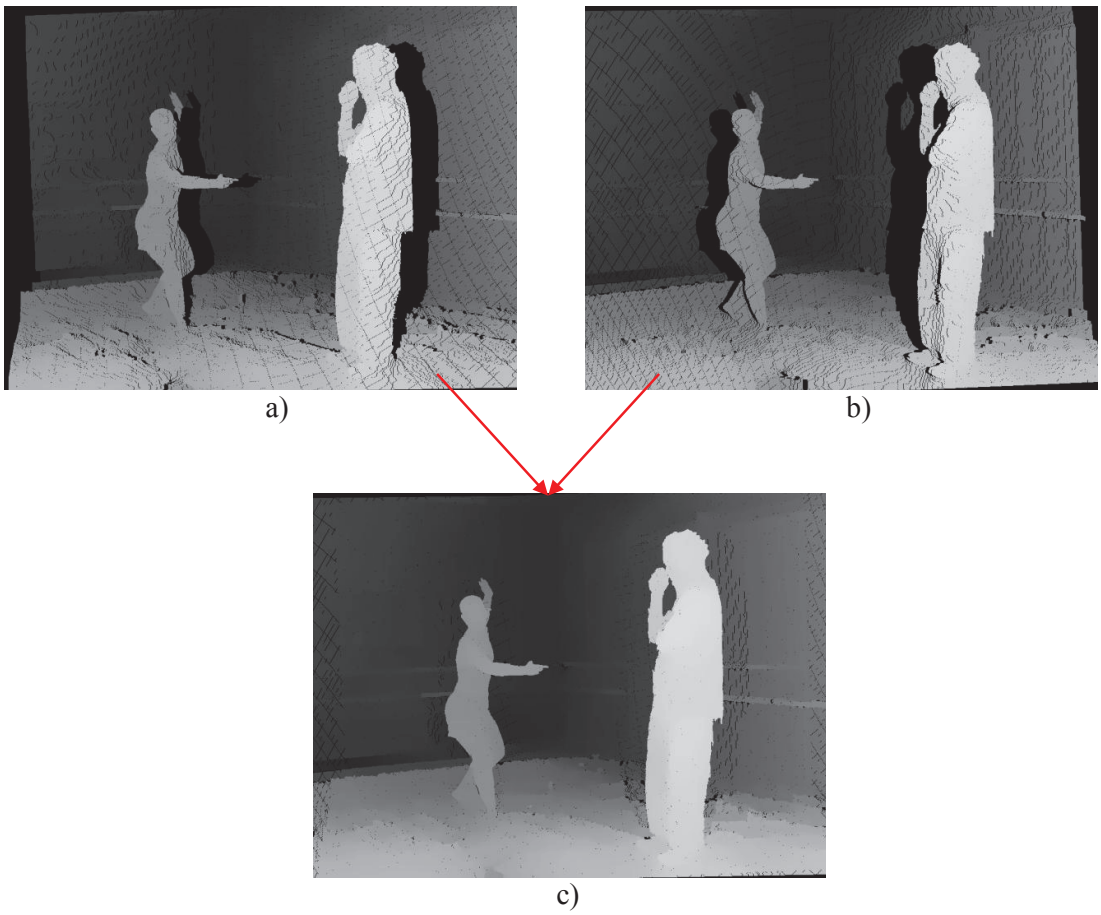


Figure 3.5: Combining depth maps : a) Warped depth left, b) Warped depth right, c) Combined depth

3.2.2 Combine Depth Maps

The warped depth maps from the left and the right cameras are blended to find the combined depth map at the interpolated position. In this way, information for the parts which are not visible to one of the cameras can be obtained from the other camera. While combining the depth value for a pixel from two cameras, the smaller pixel value, representing objects closer to the foreground, is selected. The output of the blending stage is illustrated in Figure 3.5.

3.2.3 Median Filtering

Due to the rounding errors caused by the calculations at the warping stage and the difference in the sampling rates between the original and the interpolated positions, the synthetic depth map at the interpolated position contains cracks as depicted in Figure 3.6a, [16] provides more information about this phenomenon. Median filtering is applied on the blended image to fix the cracked regions, as shown in Figure 3.6b. In addition,

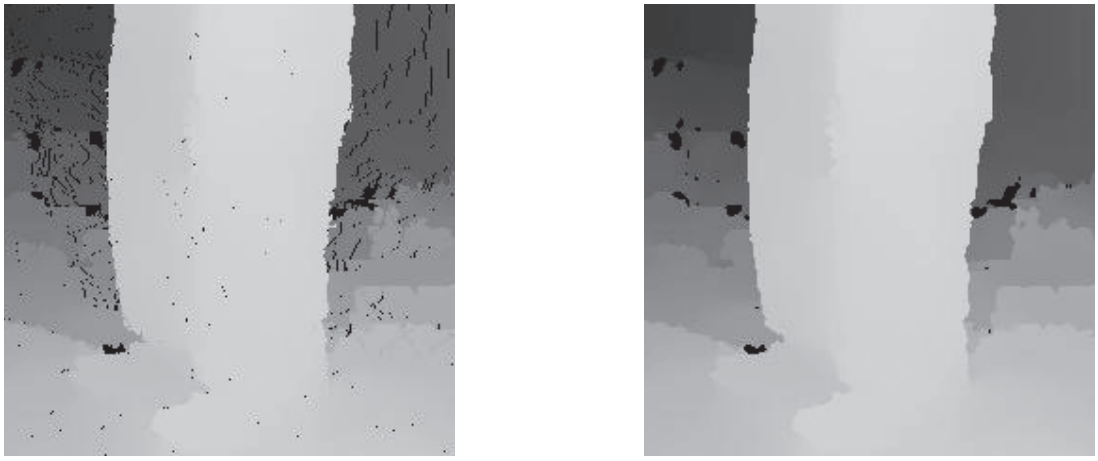


Figure 3.6: Median filtering : a) Combined depth, b) Depth after median filtering

median filtering is useful in smoothening out the depth values inside the same object while preserving the edges of different objects [17]. Furthermore, a median filter with a small window size of 3x3 leaves the dissoccluded parts of the image unchanged since these parts usually occupy clusters with a large area.

3.2.4 Disocclusion Inpainting for Depth Maps

The synthetic depth map at the interpolated position may have some regions of black pixels after median filtering. The reason for these holes is the absence of information for that pixel from both the cameras. Inpainting is used to fill the dissoccluded regions with information extracted from the neighboring image pixels which are not dissoccluded. Inpainting is performed by finding for every dissoccluded pixel the closest neighbor with a non-zero depth. Searches in horizontal, vertical and diagonal directions result in 8 neighbors out of which the one with the greatest depth value is selected, assuming that the dissoccluded pixels belong to the background. The result of the inpainting stage is given in Figure 3.7.

3.2.5 Inverse Warping Texture Maps

The interpolated depth map, obtained after the post-filtering steps described above, is used together with the camera texture views to create a virtual texture view at the interpolated position. Figure 3.8a and b demonstrate the inverse warped textures from the original left and right camera views, respectively.

3.2.6 Dilation

Due to inaccuracy in depth maps, textures are warped to incorrect places at the interpolated viewpoint. This situation is mostly visible at the edges of objects where the depth

Figure 3.7: Disocclusion inpainting for depth maps : a) Depth after median filtering, b) Depth after filling disocclusions

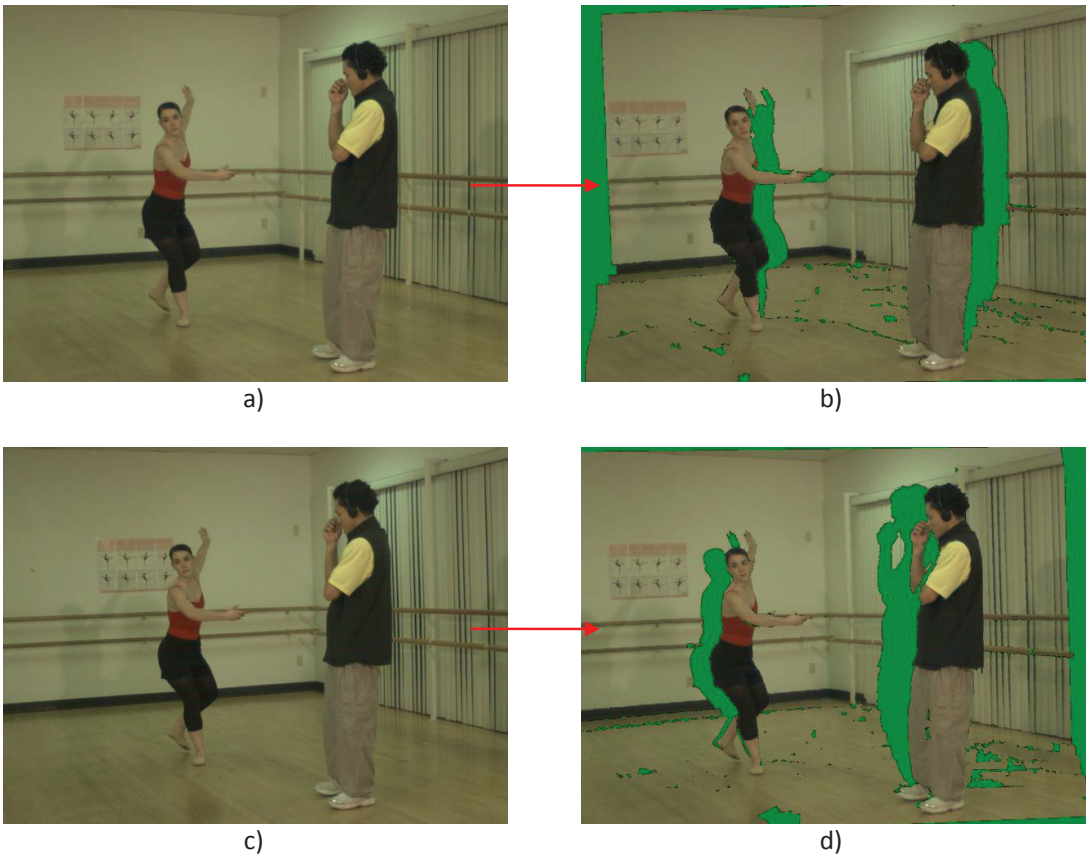


Figure 3.8: Inverse warping texture maps : a) Original left camera texture, b) Warped texture left, c) Original right camera texture, d) Warped texture right

Figure 3.9: Dilation : a) Ghost contours in the absence of dilation, b) Ghost contours erased by dilation

map shows high discontinuities [16] as opposed to the texture maps where the transitions in the edges are distributed over a few pixels. This issue causes a ghost contour of the foreground objects to appear in the background, around the border of the dissoccluded areas, as visible in Figure 3.9a. This is avoided by applying a dilation operation, which expands the dissoccluded area, before the blending step. The result of blending, preceded by dilation is given in Figure 3.9b.

3.2.7 Blending Texture Maps

The resulting synthetic views from the left and the right cameras at the interpolated position are blended using a weighted average method. The weights are based on the distance from the left and the right camera positions to the interpolated position. Figure 3.10 shows the output of this stage.

3.2.8 Disocclusion Inpainting for Texture Maps

As in the case of the projected interpolated depth map, the blended texture map might also have areas that cannot be viewed from any of the reference cameras. Assuming, again, that the dissoccluded pixels belong to the background, the depth information is also used [16] to fill the disocclusions accurately. For inpainting of a dissoccluded pixel, searches are performed in 8 directions to find the nearest pixel that is not disoccluded. A weighted average of these pixels is taken in order to calculate the value that needs to be used to fill the disocclusion. The final result which demonstrates the output view at the interpolated position is given in Figure 3.11.

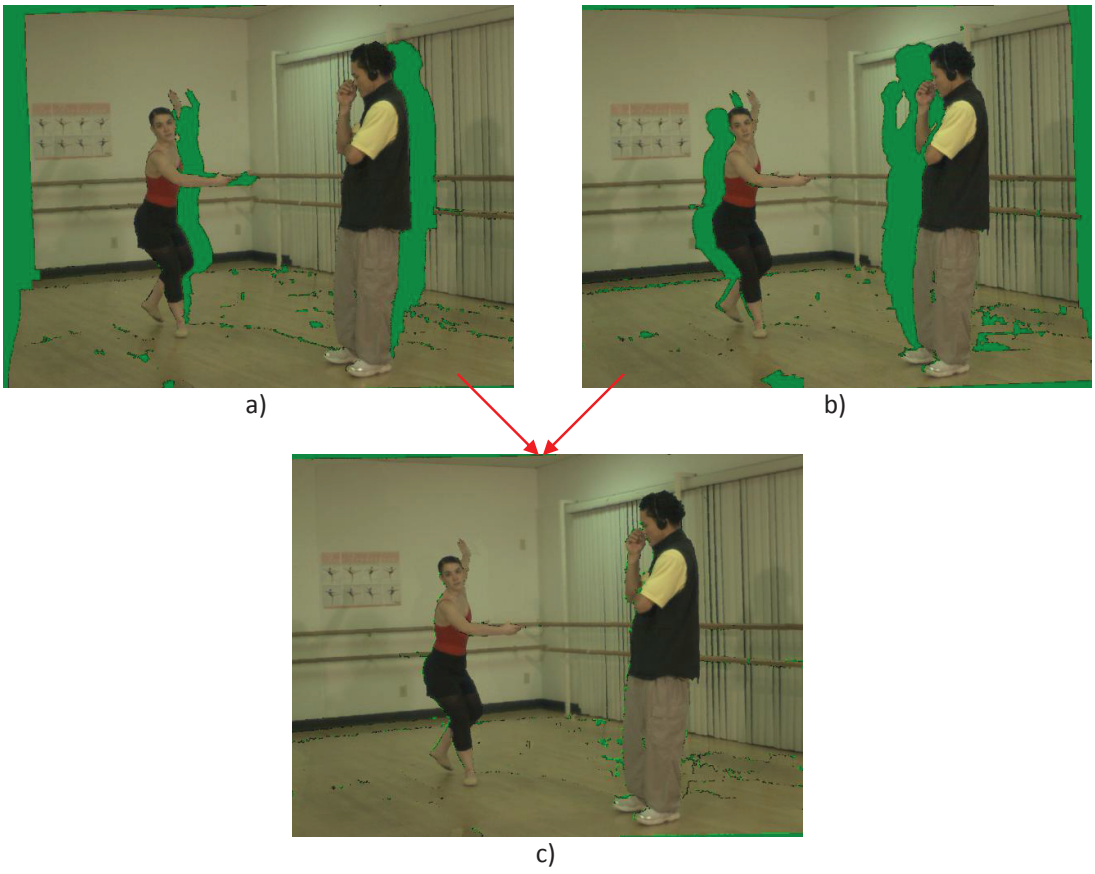


Figure 3.10: Blending texture maps : a) Warped texture left, b) Warped texture right, c) Blended texture map

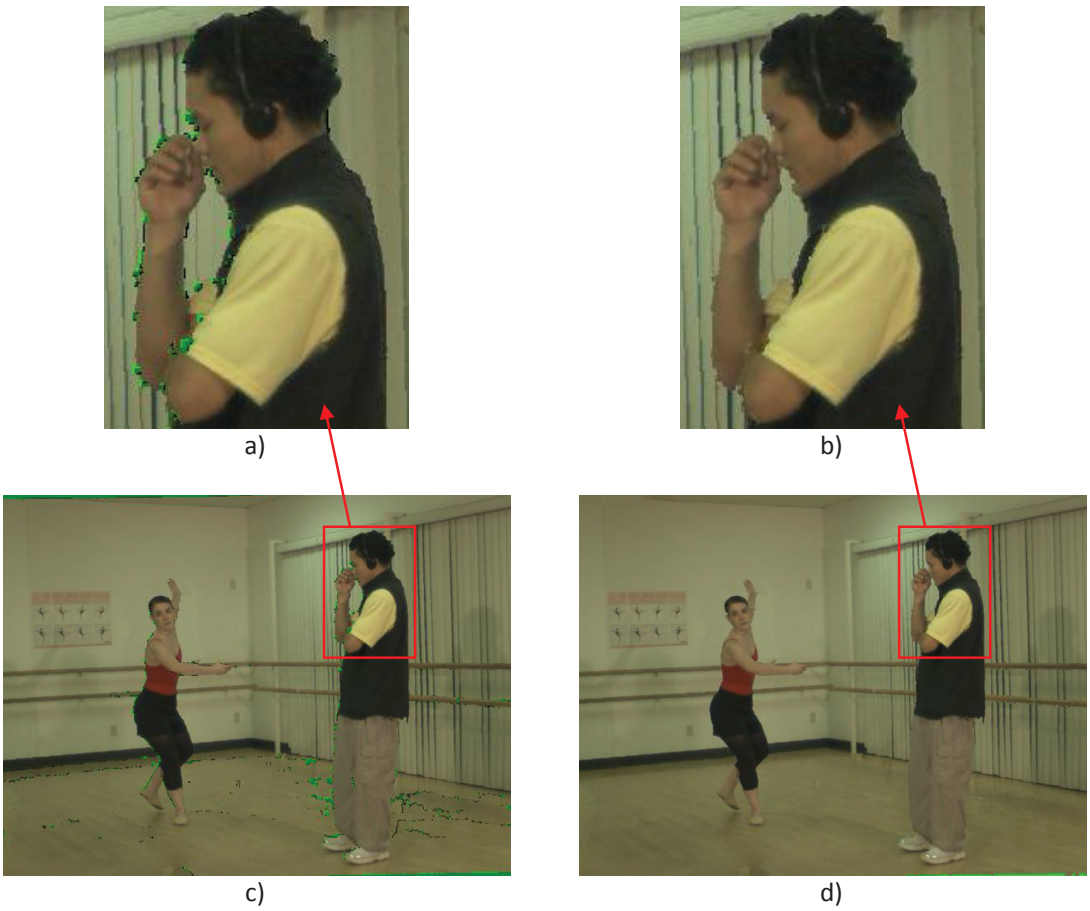


Figure 3.11: Disocclusion inpainting for texture maps : a) Before inpainting (closer look), b) After inpainting (closer look), c) Before inpainting, d) After inpainting - final interpolated view

4 FTV Rendering Platform

This chapter is dedicated to introduce the Silicon Hive ISP2400 processor based architecture on which the FTV algorithm is mapped. The target platform for rendering (iGLANCE backend board) is explained. Then, we give an overview about the individual components of the rendering platform; we also present the synchronization method for these system components, and the system-level performance estimations.

4.1 Target Rendering Platform

As explained in Chapter 2, the architecture developed in the iGLANCE project is partitioned into two boards. Silicon Hive's Gladiator board is used for the back-end platform which is responsible for FTV rendering. The Gladiator board has been used inside Silicon Hive to demonstrate various camera solutions. The development cycle for FTV rendering platform is accelerated using this verified system.

A simplified view of the Gladiator board is given in Figure 4.1. The Gladiator involves a Cirrus host board equipped with a host processor, ARM. There are two Xilinx Virtex-5 FPGAs connected to each other through 350 single-ended connections. It also includes a smaller configuration FPGA, Xilinx Spartan 3A, controlling the two application FPGAs and the rest of the system. To configure the application FPGAs, the configuration FPGA should be initialized using a platform flash or through JTAG. Once initialized, the configuration FPGA functions as a memory-mapped device for the host processor. Through this interface, the configuration data for the application FPGAs are transferred by the host processor.

FPGA-1 receives the HDMI data via an HDMI receiver chip (AD9398) placed on the board. FPGA-2 has an interface with an HDMI transmitter chip (AD9889b) to send out the HDMI data. Each application FPGA has two banks of 32-bit DDR2 SDRAM with 2 Gbit capacity. The board also contains a camera module interface, a 1024x768 TFT LCD display, an Ethernet interface and USB ports.

4.2 FTV System View

The system in Figure 4.2 is proposed for the rendering platform. The main components of the system are: host, FTV processor, external memories, control bus, memory bus, DMA, and HDMI interfaces, which are described in more detail in the subsequent sections. The system is generated from a single HSD description, in which all the blocks are instantiated and parameterized.

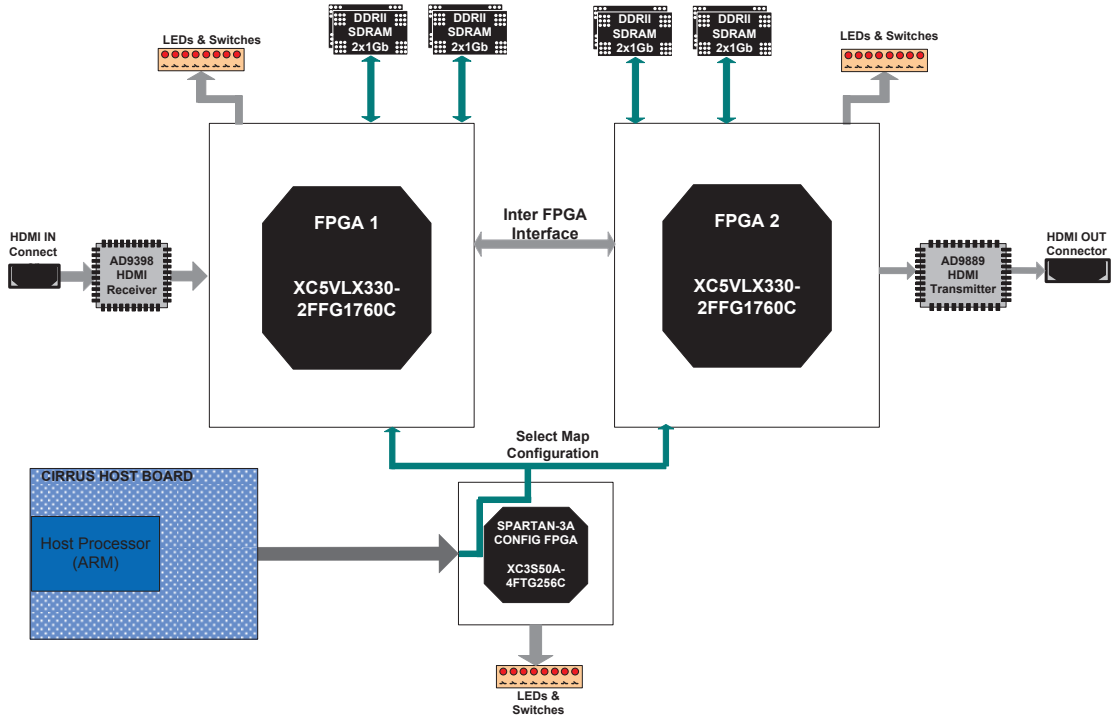


Figure 4.1: The Gladiator FPGA board

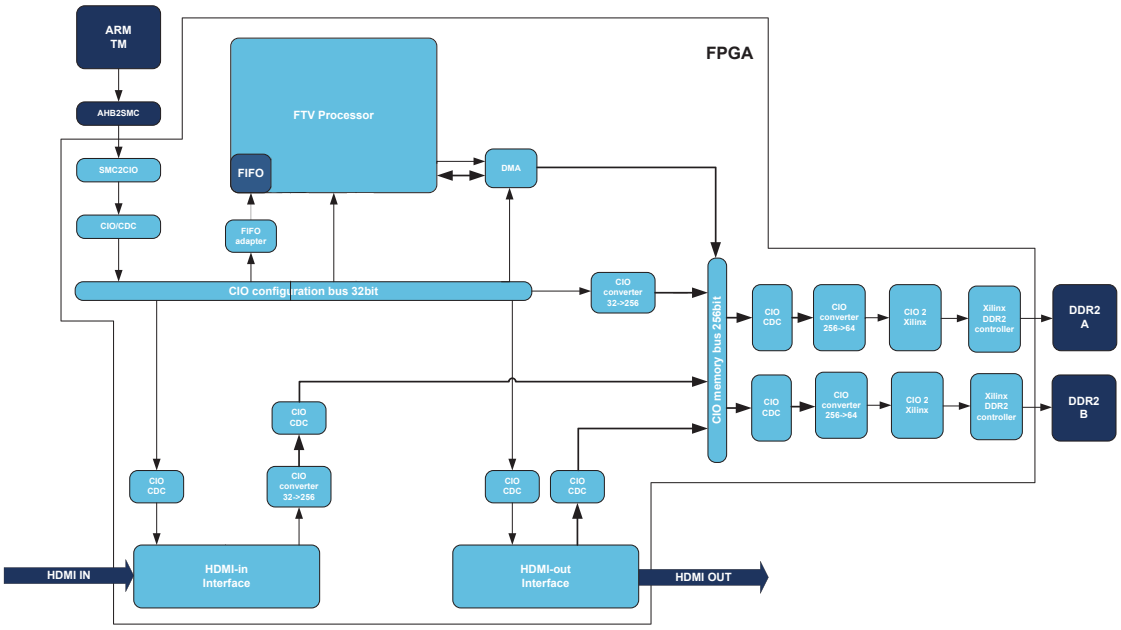


Figure 4.2: FTV system view

4.2.1 Host

The host processor is responsible for controlling the system. It has master access to the 32-bit CIO control bus, meaning that it can initiate data transfers on this bus to access the FTV processor's memories, the HDMI interfaces, and the external memory. The ARM processor on the Cirrus board is used as the host processor in the FTV system. The AMBA High-Performance BUS (AHB) protocol of the ARM processor is connected to the FPGAs through an SRAM Memory Controller (SMC) interface. In the FTV system, the conversion from SMC to the Core I/O (CIO) bus protocol takes place. Since the host processor and the FTV system running on FPGA operate at different frequencies, a Clock Domain Crossing (CDC) FIFO is needed to ensure reliable sampling of data across clock domains.

The host processor runs the control partition of the FTV algorithm by providing the kernel (the FTV processor) with the parameters which are necessary for free viewpoint interpolation. The control over the kernel is carried out by a group of layers (see Figure 4.3).

- **HRT-API:** It allows controlling the Silicon Hive processors and the other system components from the host using C function calls.
- **FTV-API:** The FTV Application Programming Interface (FTV-API), designed by [6], is an interface which allows the application level software (called the iGLANCE Middleware, see Chapter 2) to control the rendering platform. It provides some functions such as init, start, pause, resume, stop, quit to initiate/interfere/stop the operation of the kernel. The FTV-API specifies a certain structure for packing the FTV algorithm parameters, which are uploaded to the FTV kernel. Furthermore, it uses HRT functions to upload to and execute the program on the FTV processor. The HRT functions are also used to store the parameters to the processor local memory.
- **The iGLANCE Middleware:** The middleware is deployed mainly on the front-end board. However, it makes function-calls to the back-end board. These functions calculate the FTV algorithm parameters based on the iGLANCE scenario, the scene information, and the viewpoint. Then, the calculated parameters are packed according to the format specified by FTV-API, and transferred to this layer. The parameters are recalculated, packed, and uploaded every time the viewpoint or the scene is changed.

The host processor is also responsible for establishing the synchronization between the middleware, the FTV processor, and the HDMI interfaces. The mechanism to achieve this task is explained in detail in Section 4.3.

4.2.2 FTV Processor

The FTV processor is the crucial system component where the free-viewpoint rendering is performed. Initial implementation is based on a floating-point VLIW processor with 5 issue slots. This is only used in order to evaluate the performance of the floating-point

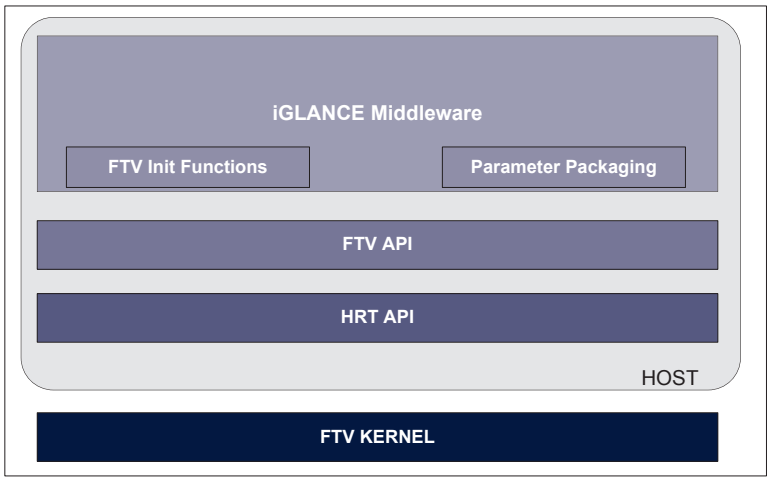


Figure 4.3: Host layered structure

Table 4.1: Performance of the out-of-the-box algorithm on a floating point VLIW core

Algorithm Stage	**Operation/pixel**	**Cycle/pixel**	**ILP**
Warp	208,02	169,71	1,23
Blend	18,3	20,65	0,91
Median	578,13	492,09	1,17
Fill Depth	36,58	27,54	1,33
Inverse warp	259,30	137,47	1,89
Dilate	92,59	87,73	1,06
Blend	157,52	63,43	2,48
Fill Y	48,63	41,14	1,18
Fill UV	11,32	9,03	1,25
TOTAL	1,410.92	1,048.79	1,35

out-of-the-box algorithm. Table 4.1 presents the results in terms of the total operation and cycle count for all stages of the algorithm.

Interpolation of intermediate views for 1080p frames at 30 fps, requires processing of 62.2 Mpixels/s. The reference implementation of the FTV algorithm indicated that a total number of 1,410 operations are required per pixel. Therefore, the corresponding throughput requirement equals to 87.7 giga-operations per second (GOPS). When this scalar processor is mapped on an FPGA running at 50 MHz, the maximum throughput is only 0.25 GOPS assuming all operations are executed in parallel.

In order to improve the performance to meet the real time constraints, an Image Signal Processor (ISP), ISP2400 is proposed as the FTV processor. This embedded C-programmable family of processors is optimized for image signal processing domain. It can efficiently perform line or pixel processing which is the case for the FTV algorithm. The ISP2400 is derived from Silicon Hive's VLIW template, and it also supports SIMD

operations. The SIMD operations can be applied on standard vectors as well as on wide vectors or vector slices. It does not have support for floating-point operations; therefore the FTV algorithm was converted to fixed-point (will be discussed in Section 4.5). As in the case of other Silicon Hive processors, it is highly configurable in the sense that the number and width of issue slots, register files, memories and types of functional units can be easily adapted.

The ISP2400-based processor (Figure 4.4), which is used in this thesis, is equipped with 7 issue slots, 2 of them are reserved for scalar operations and 5 for vector operations. Each issue slot has its own multiple scalar/vector register files. SIMD operations are 16-way, i.e. a vector consists of 16 elements. The element precision was chosen as 18-bits in order to benefit from the 18-bit multipliers on the FPGA. The processor has a program memory with a width that is equal to the length of the very-large-instruction-word. It uses scalar and vector data memories each of which are connected to a single load/store unit. Furthermore, it also includes a vector-addressable memory to support scatter/gather type of vector load/stores.

4.2.3 DMA

The DMA reads/writes data from/to the external memory based on the requests issued by the FTV processor. The processor makes the request through the communication FIFO between them. The DMA sends back an acknowledge token through the same FIFO interface following the completion of each command. The commands can be pipelined, i.e. the processor can make as many requests as the FIFO depth without having to wait for acknowledge.

The DMA block used in Silicon Hive allows configuration of various parameters in the software. The element precision for the read and write accesses is configurable, i.e. the DMA handles padding/discarding a part of the data according to the element precisions on both sides of the transfer. The DMA supports 1D and 2D communication transactions, whose sizes are also programmable. It also has configurable channels which allow setting a number of separate channels between different memory elements.

In our system, the DMA has an interface with three components, namely the external memory, the FTV processor, and the system bus. Therefore, it is capable of initiating data transfer between any of these two. The interface with the FTV processor is used to reach the data in the vector memory; therefore this connection is 288 bits-wide. The system bus connection is used to reach the other processor memories: the data memory and the vector-addressable memory. The DMA accesses the external memory through the memory bus.

4.2.4 External Memory

Ideally, the external memory should only be used to store the initial input and the final output of the algorithm. The HDMI-in interface, which receives the decoded video stream from the front-end board, writes this data into the external memory in a planar format. The FTV processor does not have direct access to the external memory but uses the DMA to receive the required section of the frame. The final output of the

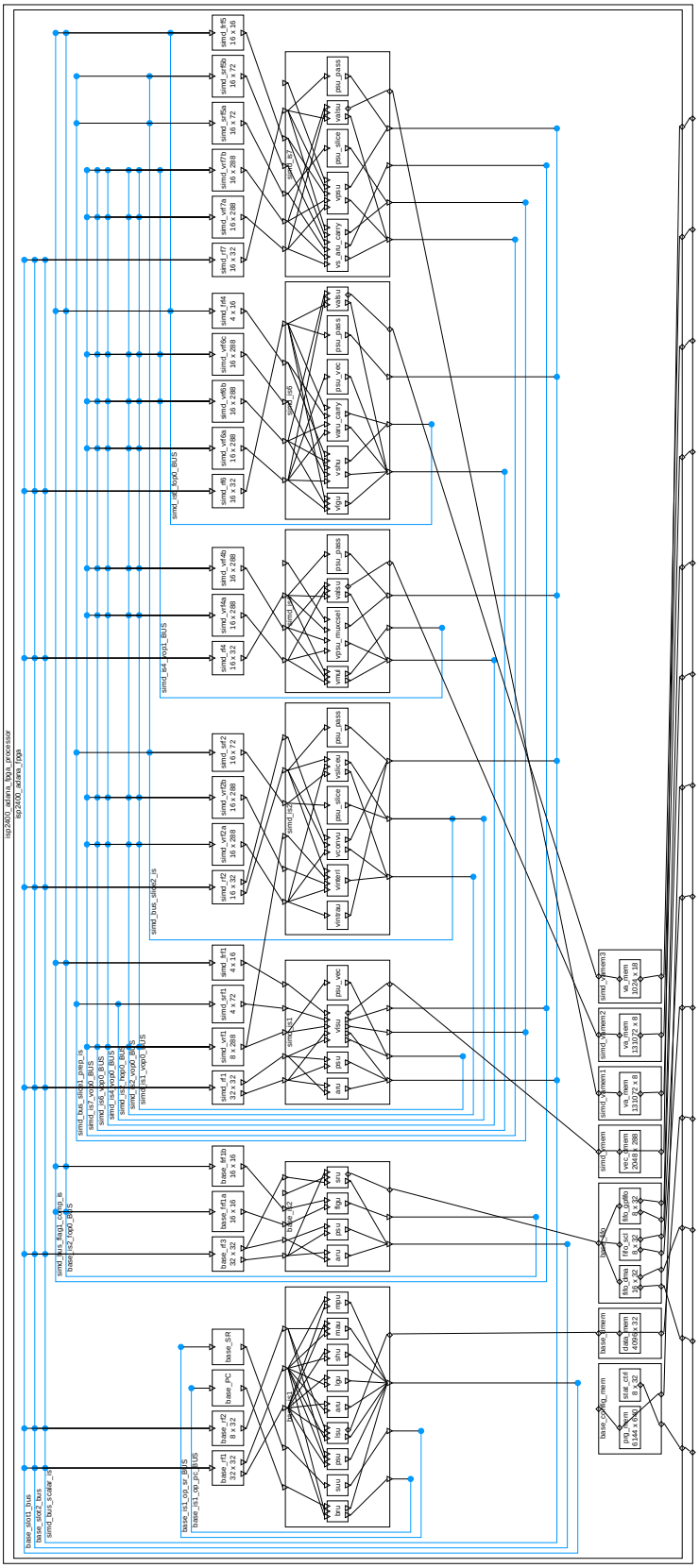


Figure 4.4: FTVProcessor

FTV algorithm is written back to the external memory using the DMA. The HDMI-out interface then reads this data and transmits it to the display.

In addition to the ideal scenario, some intermediate results of the algorithm are also stored in the external memory. This is because some stages of the algorithm are not merged in the context of this thesis. Due to the limited amount of processor local memory, the result of every stage, which is not merged with the next function, needs to be stored in the external memory.

4.2.5 HDMI Interfaces

The HDMI-out interface reads the output data from the external memory and sends it to the display module through the transmitter chip on the Gladiator board. The HDMI-out interface is controlled by the host processor by configuring the frame format and size information as well as the memory address from which the data is read. Once enabled by the host, it keeps sending the frames from the specified read address. This address is updated in the host application code to alter the displayed frame. This block was present in Silicon Hive device inventory, it was tested and ready-to-use.

The HDMI-in interface captures the incoming data stream from the HDMI-Receiver (Rx) chip on the Gladiator board, partitions it into separate planes, and writes it into the external memory. The configuration of the HDMI-in interface is also handled by the host. Figure 4.5 illustrates the block diagram of the interface. It is divided into two partitions, being control-related and data-related.

- **Control-related Blocks:** The control blocks are *CIO-slave* and *I2C-master*. The host processor commands, sent through the CIO configuration bus, are received by the CIO-slave block, and these commands set a number of registers which are then used to send control signals to the other blocks. The host can also read the status by reading from a number of reserved registers. The I2C-master block is responsible for the configuration of the Rx chip using the I2C protocol.

- **Data-related Blocks:** The rest of the blocks form the data partition. *Capture-data* receives a new frame from the RX chip. It samples the data using the data-clock (DATACK) generated by this chip. Therefore, a CDC FIFO is needed to safely transfer the data further. *Partition-data* block divides the pixels into Y, U, V, and Depth planes by storing them into separate FIFOs according to the color format information. Currently, only RGB and YUV444 formats are supported. These FIFOs are successively read by the CIO-master block forming 32 bit (4 pixels) data belonging to the same color component. After this, the gathered pixels are sent to the external memory via burst requests.

An initial version of the HDMI-in device was described in [6]. However, it did not support all the iGLANCE scenarios. Therefore, a number of modifications were required. These modifications are listed here:

1. **Functionality:** The initial design was modified to synchronize capture-start with both the enable and the VSYNC signals.

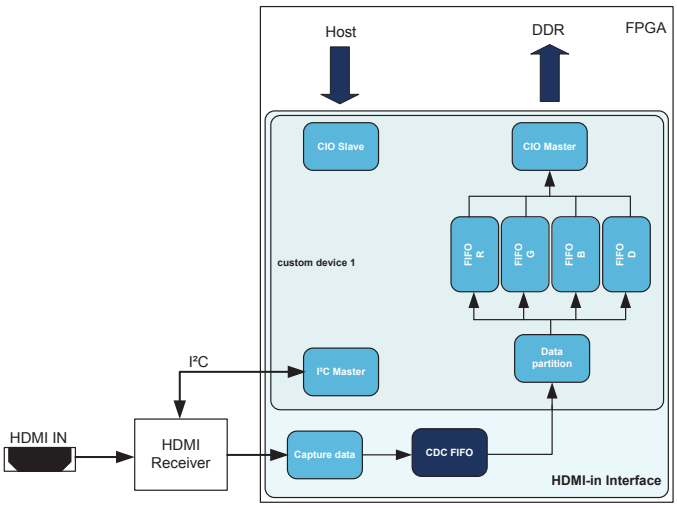


Figure 4.5: HDMI-in interface

2. **Improved SW Programmability:** Previous design was based on hard-coded values in I2C-Master block to configure the HDMI-Rx chip. This required re-synthesizing the system every time when we need to change the configuration. We extended CIO-Slave with additional control registers and added the I2C-master a state-machine to program the HDMI-Rx. In this way, the programmer can configure the HDMI-Rx chip using the C-code. The values in the CIO-Slave registers are set by the host application; and the state machine in I2C master gets triggered to configure the receiver chip based on those register values.

3. **Software Reset:** A software-reset signal is introduced in order to reset all the register values in the HDMI-in device without having to restart the Gladiator board. This is achieved by reserving a register in the CIO-slave block. Upon setting this register by the host application, a soft-reset signal is sent to all the sub-blocks. To achieve this, we modified their corresponding VHDL source codes. However, the CDC FIFO is a Silicon Hive standard device, and its reset port is invisible to the designer in the HSD system description. We solved this problem by flushing this FIFO when the software reset is asserted, i.e. reading all the data, and discarding the values.

4. **Handling Watermarking:** In order to support the format required by the iGLANCE consumer-scenario (see Chapter 2), the HDMI-in module was upgraded such that it can receive the camera views from the left and the right angle, and store them into separate buffers in the memory. In the context of the iGLANCE project, the first 8 pixels of the left and right camera frames are watermarked with white and black pixels, respectively. The modified HDMI-in interface can now use the watermarking information and upload the received frame into the corresponding memory location (Figure 4.6). Therefore, since the check is done in hardware, the software does not have to perform memory reads to check watermarking.

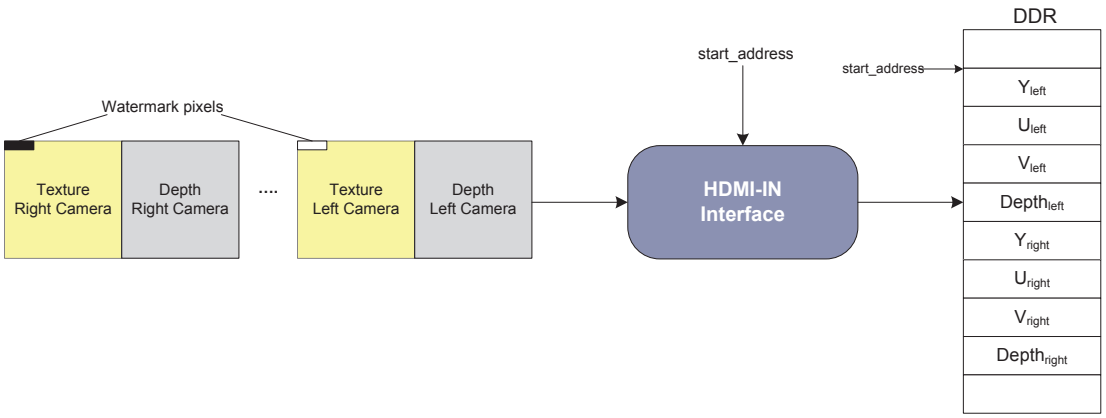


Figure 4.6: Watermark support in HDMI-in interface

4.3 System Level Synchronization

The desired method was to implement all synchronization elements in hardware. However, given the lack of time, the synchronization between all the processes is handled by the host. On one side, the host should take the required action upon a command received from the front-end board. This requires polling the proxy port for incoming remote-procedure calls continuously. On the other side, the host must ensure that the HDMI interfaces and the FTV processor are working on separate buffers in the external memory. Both these processes require endless loop structures. In order to ensure that the two continuous processes can run together and communicate with each other, the host is executing two parallel threads using Posix Threads (Pthreads) libraries (see the code in Figure 4.7). Shared variables between the two processes provide the means of communication.

```
int hrt_main(int argc, char **argv)
{
   pthread_t    thread1, thread2;

   if(pthread_create(&thread1, NULL, (void *)iG_main_loop, NULL))
      printf("Error in creating thread2\n");

   if(pthread_create(&thread2, NULL, (void *)IGFORPre_rpc_stub_main, NULL))
       printf("Error in creating thread1\n");

   while(1)
      sleep(1);
}
```

Figure 4.7: Creating two threads for handling synchronization

The first thread executes the process called iG_main_loop which handles the synchronization of the FTV processor with the HDMI interfaces (see Figure 4.8). Its first task is

to initialize both HDMI-in and out interfaces by configuring their registers. After that, they are enabled with initial start addresses (InBuf#0, OutBuf#1). Then the main loop starts with the host waiting until two frames (left and right) are received by the HDMI-in interface. When both are received, the start address for the HDMI-in is updated, and it is enabled again so that it can write the next two frames to the second buffer location (InBuf#1). While the HDMI-in is writing to this buffer, the FTV processor can start processing the views in Inbuf#0 unless there is an interrupt from the RPC thread. This interrupt occurs only when there are new parameters to be uploaded to the FTV processor. If there is an interrupt, the first thread waits for the second thread to finish uploading. After running the FTV processor for the corresponding input and output buffer addresses, the addresses of the HDMI-out interface is updated to read from the location that is filled by the FTV processor. The procedure for the rest of the loop is similar; the buffer that is used by each device is altered as shown in Figure 4.9. While this thread checks the *RPC_INTERRUPT* variable from the second thread, it also sets the variable *FTV_STATUS*. *FTV_STATUS* is assigned to *BUSY* before running the algorithm on FTV processor, and it is assigned back to *IDLE* when the processor is done processing.

The second thread waits for the RPC calls from the front end board. When it receives a call, it branches to the corresponding function. If the called function requires uploading new viewpoint/dataset parameters to the kernel, it waits until the kernel is IDLE by checking *FTV_STATUS* variable. Meanwhile, it also sets the variable *RPC_INTERRUPT* to *TRUE* so that the first thread stalls until the upload of the new parameters is completed.

While talking about threads accessing a shared memory location, it must be ensured that these accesses are safe, i.e. free of race conditions. A race condition occurs when multiple threads concurrently read and write to a shared location and the result depends on the order of execution. If these pieces of codes, called critical sections, are not handled properly, it might cause a deadlock, in which threads wait indefinitely without any progress. In order to prevent this, mutual exclusion principle is used to restrain the ways the requests are made. The *pthreads* library provides *mutex* variables for this purpose [8]. It behaves like a lock giving only one thread the permission to access a shared resource. In our case the global variables *RPC_INTERRUPT* and *FTV_STATUS* are shared by the two threads using mutual exclusion principle.

4.4 Performance Estimations

The requirement for the iGLANCE consumer scenario is specified as 30 Full-HD (1920x1080) frames per second, each frame is packed in the YUV 4:2:0 format. The frequency for capturing 1920x1080p frame sequence is specified as 148.5 MHz by the CEA-861-D high speed digital video standard [10].Due to the limitations of the HDMI-in interface which is not functional above 120 MHz, this requirement is downscaled to HD-ready (1280x720p) frames packed in YUV 4:4:4 running at 30 fps for demonstration purposes. The standard specifies the frequency as 74.250 MHz for streaming in 1280x720p resolution at 50 Hz refresh rate; therefore the current system can correctly capture the streams in this format.

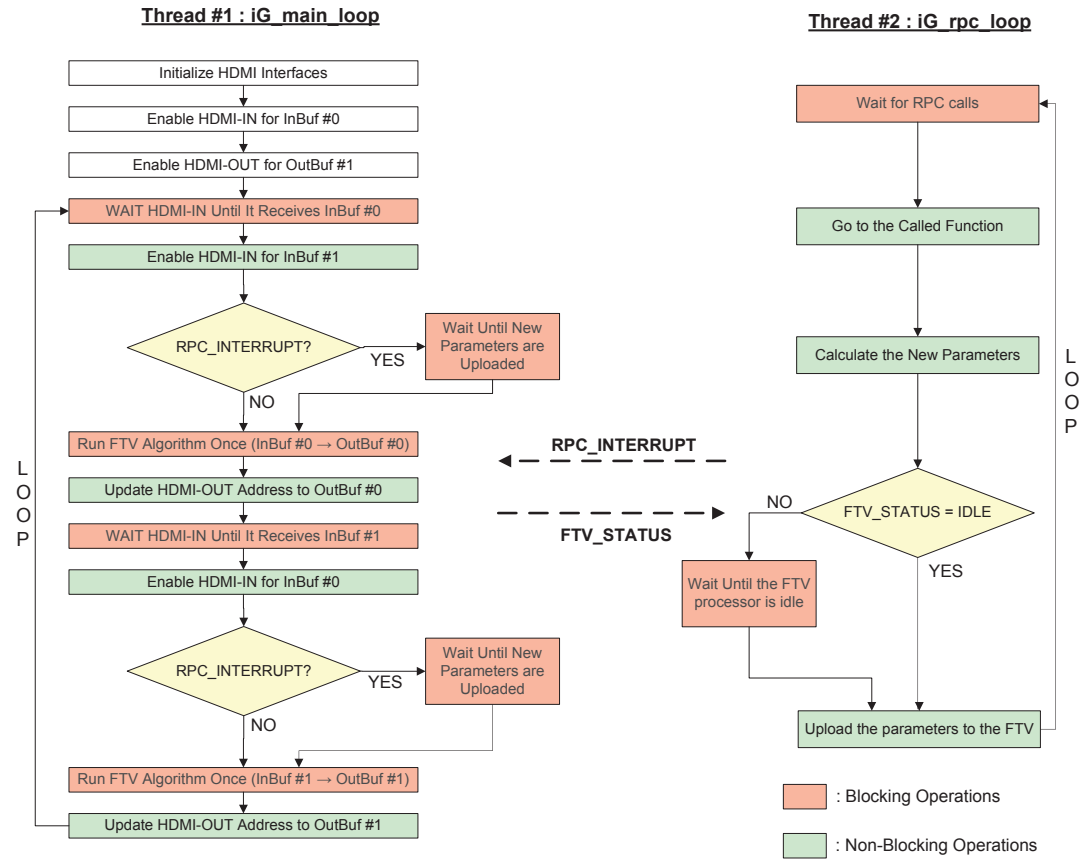


Figure 4.8: FTV system synchronization

Figure 4.9: Double buffering scheme

4.4.1 Throughput Requirement

The throughput requirements for both scenarios are demonstrated in Table 4.2. Downscaling to a 1280x720 resolution reduced the necessary processing power from 87.7 GOPS to 39 GOPS, a factor of 2.25. In order to reach such a high processing power, the advantages of both instruction-level and data-level parallelism should be exploited using a VLIW processor with vector data paths.

Table 4.2: Throughput requirement

Scenario	Mpixels/s	GOPS
1080p @ 30fps	62.2	87.7
720p @ 30fps	27.6	39.0

4.4.2 Memory Bus Bandwidth

In order to calculate the memory bus utilization, the contributing factors from all system components are accumulated. Table 4.3 presents these calculations.

Although the depth values are represented by an 8-bit value, the depth information in the incoming stream is not packed, and represented with 24-bits. In order to reduce redundant bus transfers, the HDMI-in device stores only the valid 8-bit information, discarding the remaining 16 bits of each depth pixel. For this reason, the estimations for depth and texture are given separately in the table.

The minimum required bandwidths to meet the real time constraints are 465 Mbytes/s and 336 Mbytes/s for 1080p and 720p scenarios respectively. Currently, the peak bandwidth of the memory bus using the 256-bit bus running at 50 MHz is equal to 1.6 Gbytes/s. Therefore, the bus utilization is 29% and 21% in the respective scenarios.

Table 4.3: Memory bandwidth calculations

Scenario	Operation	Mpixels/s (Texture+Depth)	Bytes/pixel (Texture+Depth)	Bus Utilization (Mbytes/s)
1080p @ 30 fps (YUV 4:2:0)	HDMI-In writing @ 60fps	62 + 62	1.5 + 1	155,00
	FTV processor reading @ 60 fps	62 + 62	1.5 + 1	155,00
	FTV processor writing @ 30 fps	31 + 31	1.5 + 1	77,50
	HDMI-out reading @ 30 fps	31 + 31	1.5 + 1	77,50
	TOTAL	**186 + 186**	**1.5 + 1**	**465 (29%)**
720p @ 30 fps (YUV 4:4:4)	HDMI-In writing @ 60fps	28 + 28	3 + 1	112,00
	FTV processor reading @ 60 fps	28 + 28	3 + 1	112,00
	FTV processor writing @ 30 fps	14 + 14	3 + 1	56,00
	HDMI-out reading @ 30 fps	14 + 14	3 + 1	56,00
	TOTAL	**84 + 84**	**3 + 1**	**336 (21%)**

4.5 Fixed Point FTV Algorithm

In order to map the algorithm to our ISP2400-based processor, which does not support floating-point operations, the floating point arithmetic should be converted to fixed-point. A fixed-point number can be thought of as an integer multiplied by a two's power with negative exponent [15]. Fixed-point numbers contain an integer part and a fractional part, with an imaginary decimal point which is placed in the same position for the same variable. The number of bits representing the integer section is called the integer word length (IWL), and the number of bits assigned to the fractional part is the fractional word length (FWL). The total word length (WL) of a fixed-point number is

therefore given by the following:

$$WL = IWL + FWL \tag{4.1}$$

The range R and the quantization step Δ depend on the IWL and FWL as follows:

$$-2^{IWL} \leqslant R < 2^{IWL} \tag{4.2}$$

$$\Delta = 2^{-FWL} \tag{4.3}$$

While performing fixed-point arithmetic, the position of the decimal point should be treated carefully. Addition and subtraction between two fixed-point numbers x and y should have the same IWL. If this is not the case, the operand with the smaller IWL should be scaled, and the IWL of the result z is therefore equal to:

$$IWL_z = max\{IWL_x, IWL_y\} \tag{4.4}$$

For the multiplication and division, the result IWL is given by,

$$IWL_z = \begin{cases} IWL_x + IWL_y & \text{for signed number} \\ IWL_x + IWL_y + 1 & \text{for unsigned number} \end{cases} \tag{4.5}$$

The FPGAs on the Gladiator board have 18-bit multipliers. In order to avoid a multiplication operation to be converted to a number of multiplications, the WL of algorithm variables is chosen as 18-bit. The conversion of the algorithm from floating-point to fixed-point was done by the iGLANCE partner TIMA. All variables in the algorithm were analyzed to find out the best possible IWL and FWL for them. Fixed-point conversion reduced the quality of the interpolated view. The quality comparison between the two is given in Chapter 6. Furthermore, the fixed-point code requires a larger number of operations than the floating-point version since a shift operation is needed after every fixed-point multiplication in order to keep the WL at 18 bits.

The FTV processor is also configured such that the element precision of a vector is 18-bits. Intermediate results of multiplications, which can exceed 18-bit, are handled using wide-vector operations. Figure 4.10 illustrates an example multiplication, where the operands, *vec0* and *vec1*, are vectors with 18-bit element precision. The resulting element values can reach up to 36-bit; therefore the required double precision is provided by a wide-vector, *wvec_temp*. The wide-vector is nothing but a pair of vectors, one holding the most significant 18-bits, *wvec_temp.hi*, and the other containing the least significant bits, *wvec_temp.lo*. These two vectors can be accessed separately if needed. In the example shown, a shift operation is performed after multiplication, which is typically the case to keep the WL at 18 bits.

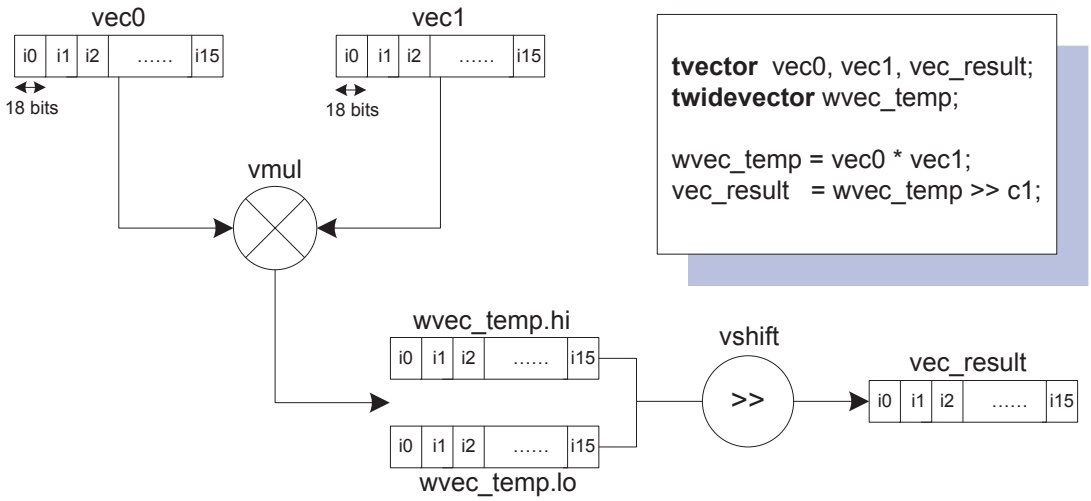


Figure 4.10: A vector multiplication resulting in a wide-vector

5 Mapping the FTV Algorithm

In this chapter, we explain the mapping of the FTV algorithm on the VLIW processor containing the SIMD datapath. The challenges related to vectorizing the algorithm, and techniques to solve these challenges are explained. Particularly, it addresses the key contribution of this thesis, which is the technique to improve the performance in the presence of irregular memory accesses. It also explains the optimizations which were applied to further improve the performance of the algorithm.

5.1 Vectorization

5.1.1 Irregular Memory Access Patterns

Background

The SIMD processors are very powerful in executing programs that contain high levels of data-parallelism as in the case of media and signal processing applications. Multiple SIMD data paths have the ability to perform a single processor instruction on aligned data elements, which are usually placed consecutively in the memory, simultaneously. Accessing all elements of a vector in a single cycle is desired to exploit the processor data paths efficiently. However, the data elements in a program are not always stored/loaded from the consecutive memory locations. In fact, there are three different ways to access the memory in SIMD architectures:

- **Unit-stride Access:** This is the simplest way in which the loads/stores are performed from/to the consecutive memory addresses.
- **Strided Access:** In this scheme, memory accesses are not directed to the consecutive locations but the addresses are spaced with a constant step called *stride*.
- **Irregular (Indexed) Access:** This refers to an access pattern where the vector elements are accessed in a random fashion usually by being indexed through another array.

An irregular data access operation causes many overhead cycles and can easily become a performance bottleneck in SIMD processors. To explain how this performance drop occurs, consider the following code segments:

$$M[i] = c \times N[i]; \tag{5.1}$$

$$M[A[i]] = c \times N[B[i]]; \tag{5.2}$$

In Example 5.1, subsequent load/store indices for arrays N and M have a constant stride. This would result in a regular memory access in which load/store operations are performed for the contiguous memory locations. Vectorizing this code is a relatively easy task. The necessary adjustment would be using vector memory which is a memory with the word size being equal to the number of vector elements multiplied by the element precision. Therefore, the whole vector can be accessed in a single cycle.

In Example 5.2, on the other hand, there is a load operation which gets data from non-contiguous locations of the array N according to the index *B[i]*. Here the index is an array (non-constant) as opposed to the previous example. Assuming that the elements of array B do not follow a uniform pattern, vectorizing this code segment would require gathering elements from different vectors. If the processor does not have the support to handle irregular accesses to memory, then sequential scalar load/stores must be performed for each vector element. This reduces the speed-up coming with vectorization, hence leading to a performance drop in SIMD architectures.

There has been extensive research on overcoming the memory access bottlenecks in SIMD architectures. Some pipelined vector processors are equipped with special hardware, a scatter-gather unit, which allows vectorization of irregular load/store operations by holding the data address sequence in a separate buffer [4]. A gather operation packs multiple data elements, which are loaded from non-contiguous memory locations, into one vector register. On the contrary, a scatter operation is performed to unpack the elements of a vector register into randomly located addresses. Another approach was to handle irregular memory accesses by employing a multi-port memory unit [25]. In [11], a dedicated hardware called a packing buffer is proposed. This, too, contains a small size multi-port memory block and addresses are given by a vector index register. A compiler-based dynamic scratch-pad memory (SPM) management scheme is proposed by [12]. The compiler is responsible for analyzing each loop and inserting the code to collect information which will be used at run-time to choose whether the Scratch-Pad Memory (SPM) should be used for a given set of irregular data accesses.

Proposed Method

In this thesis, a similar scatter/gather unit to the ones mentioned above is used. Silicon Hive ISP family processors already have a functional unit, *valsu* (vector-addressable load-store unit), to realize such operations. This functional unit is connected to a special memory called *vamem* (vector-addressable memory) and has two basic operations, namely *OP_vmldo* (see Figure 5.1) and *OP_vmsto*. A scalar register, base address, and a vector index register (*offsets*) are the inputs for the *OP_vmldo* operation. The physical address of each vector element to be loaded is calculated by adding the offset in the base register to the index register. This operation returns the resulting vector holding the value of each vector element at the requested addresses. The store operation (*OP_vmsto*) works with the same principle except that it requires a third input, vec_values, which contains the vector of values to be stored at the indexed addresses of the memory.

As proposed in the above mentioned research, two different approaches are applied along with scatter-gather units. In the presence of multi-port memories, the load/store of a vector of elements takes a single cycle if the number of memory ports is equal to

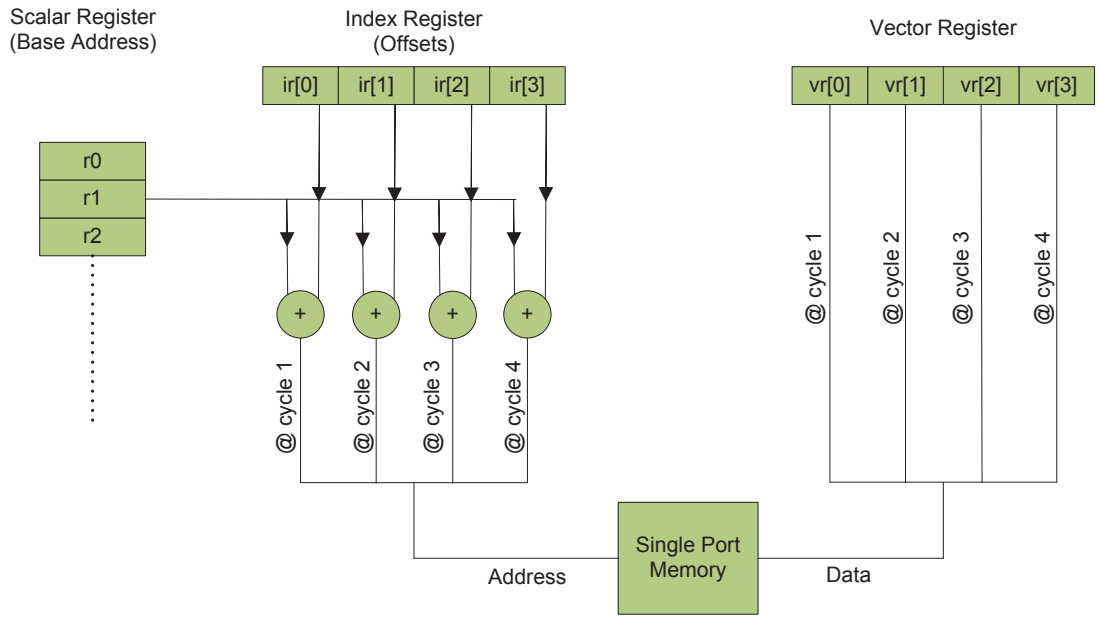


Figure 5.1: An example of a vector-addressed load operation

the number of vector elements (*Nelems*). Multi-port memories used in [11] and [25] have very small capacities since using a multi-port memory can be quite costly in terms of chip area. The use of single port memories, on the other hand, is cheaper while each vector load/store operation takes *Nelems* cycles. Although the execution latency is the same as in the architecture without the scatter-gather unit, it still has the advantage of maintaining continuity in the execution pipeline of the processor. Once the processor issues the operation, it continues issuing other operations while the scatter-gather unit handles *Nelems* separate load/store requests to the memory.

In this work, the vector-addressable memory is used as a data-cache, to hold the high-resolution frame; therefore the required memory size is very large. The target platform of the iGLANCE project, however, is an FPGA board in which the total available memory size is limited. Furthermore, there are a lot of computations in the functions in which random memory accesses occur. Therefore, the latency introduced by the use of a single port memory as *vamem* can be effectively hidden in these functions. This led us to select a single-port *vamem* and use it together with the scatter/gather unit.

In the FTV algorithm, warping of depth maps and inverse warping of texture maps are the steps in which irregular memory accesses are performed. In these steps, based on the value of the pixel as well as the camera and viewpoint parameters, the new coordinates of that pixel in the resulting depth/texture image are calculated. As the nature of this algorithm step requires, depending on the pixel depth value, elements (pixels) of a vector might scatter into a different vector inside the image making regular vector load/store operations impossible. Figure 5.3 shows this phenomenon. A vector of 8 elements in the input depth map is warped to four different vectors in the resulting depth map after warping.

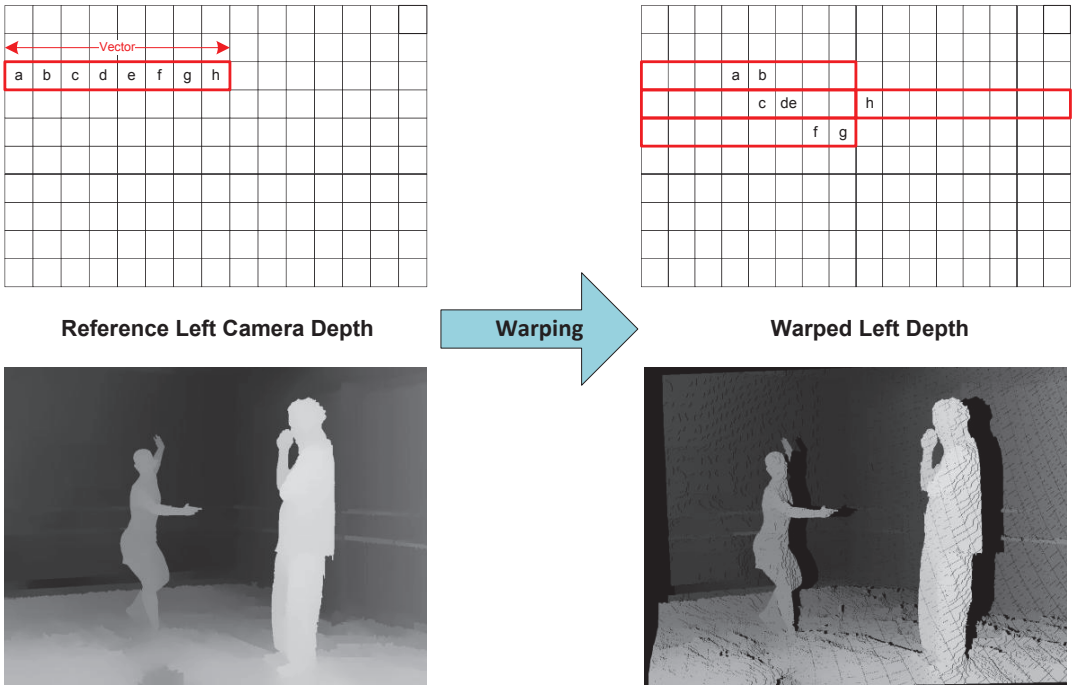
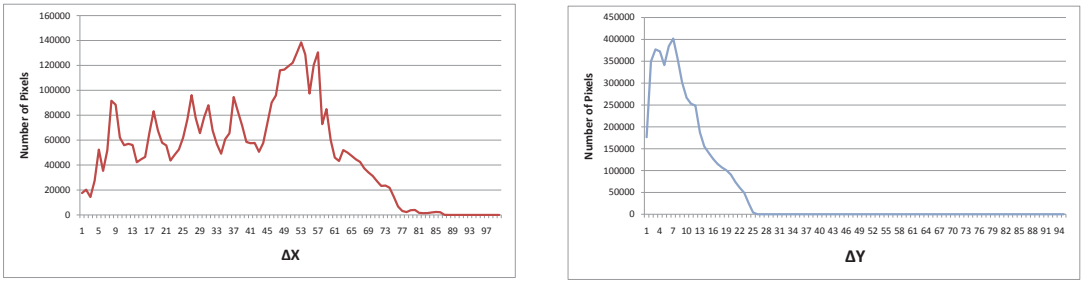


Figure 5.2: Data scattering at the warping stage

With the use of *valsu* in combination with *vamem*, the problem of the warping stage can be solved. The address of the top-left pixel should be given as the base offset, and the calculated coordinate value of the pixel is given as an index vector. However, this would typically require storing the whole output frame in the processor local memory (*vamem*). The FPGA that is used in this project is a Xilinx Virtex-5 XC5VLX330. The maximum available block-RAM size on this device is 10,368 Kbits. Storing even a single full-HD frame in YUV 4:2:0 format, on the other hand, would require 24,883 Kbits. Therefore, the proposed solution is to store a part of the frame in the scratch-pad memory, *vamem*, while the whole frame is located in the external memory. The following sections explain the way to choose the size of this memory and a smart mechanism to provide the flow of data between the two memories.

Analysis on the Required Local Memory Size

In order to choose the size of the required local memory, an analysis was conducted on the warping stage of the algorithm. The amount of scattering in pixel coordinates is determined for various viewpoint positions. The sample video is the ballet stream provided by Microsoft Research with 1024×768 resolution and the eight camera positions as explained in Chapter 2. Table 5.1 indicates the maximum absolute scattering in x and y-directions for different viewpoints. The first row, for instance, represents the interpolation for the viewpoint at camera position-1 implying that camera-0 and camera-2 are used as the left and right cameras. Since the distance between the left/right camera



(a) (b)

Figure 5.3: The amount of pixel scattering at the warping stage

and the viewpoint position is the maximum possible value (4°), this table gives a clear idea about the maximum range that we need to cover in vamem for this dataset.

Figure 5.3(a) shows a plot between ΔX and the number of pixels which are mapped by a distance of ΔX, where ΔX is the absolute difference between the pixel's original x-coordinate and its warped x-coordinate. Figure 5.3(b) depicts the corresponding plot for the y-coordinate. These plots are obtained by accumulating the results from various viewpoint positions. This also suggests that the data in the cache can be limited to a certain range for this specific dataset.

Table 5.1: The maximum amount of scattering for different viewpoints

Viewpoint Position	ΔX_{max}	ΔY_{max}
1^{st} camera	79	20
2^{nd} camera	75	19
3^{rd} camera	75	19
4^{th} camera	72	26
5^{th} camera	86	20
6^{th} camera	63	29
Global Max	**86**	**29**

Based on the results of this analysis, reserving a window consisting of $(2 \times \Delta X_{max}) \times (2 \times \Delta Y_{max})$ pixels of the output frame in vamem is sufficient to guarantee that the warped pixel coordinates will be covered. In such a case, the image is scanned in row by row fashion while maximally reusing the shared pixels. More precisely, the window is shifted to right through the first row, and it goes down by one row in the same column, then the second row is scanned towards the left direction, and so on. There are, however, problems with this scheme which are listed as follows:

- After each shift of the window, the column that is processed should be transferred to the external memory via DMA. This column-based transfer is costly since the pixels of a column are not in contiguous memory locations making burst transfers impossible.
- After the completion of processing a line, the window shifts to the line below. Then,

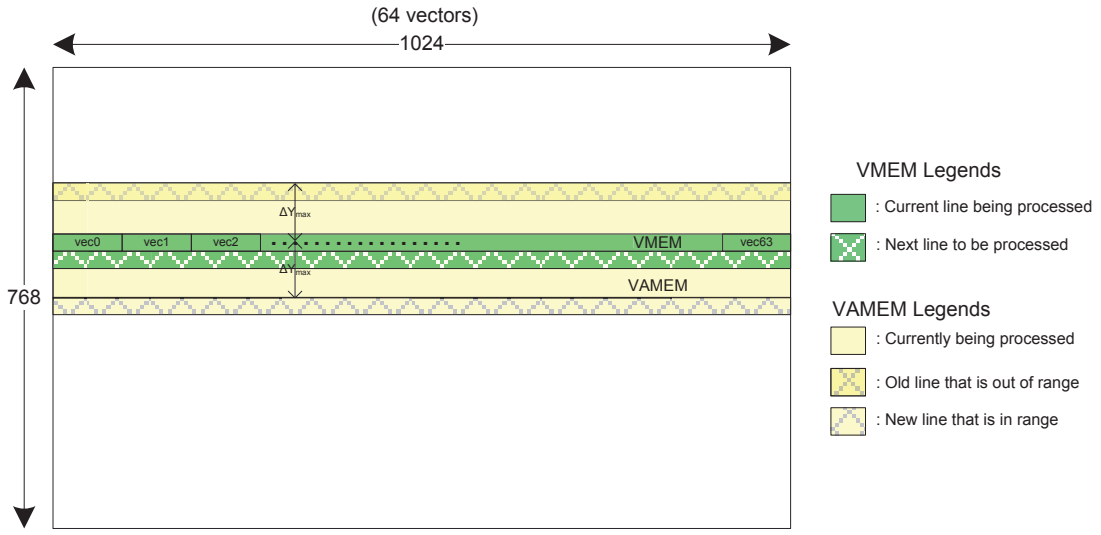


Figure 5.4: The proposed method for caching data in the local memories

all pixels in the column (except for the top one pixel only) which were sent to the external memory can be updated again, therefore they are required to be placed into the local memory again. This incurs many redundant load/store operations.

In order to avoid these problems, a better alternative is to adopt the scheme shown in Figure 5.4. It should be noted that the vmem holds the input lines while the vamem is reserved for the output lines, although they are depicted in the same frame in this figure. The block which needs to be stored in the vamem consists of $2 \times \Delta Y_{max}$ output lines of the frame. Despite a larger memory requirement compared to window-based solution, data transfer between memories can be arranged in a more efficient way. This method guarantees that once a line of pixels is transferred to the external memory it will not be updated again, meaning that there is no redundant memory transfer. The details of this technique regarding the way to use the vector and vector-addressable memories are explained in the following subsections.

The use of vector memory

Vector memory (*vmem*) is used to hold the current line of the input depth map that is being processed. In order to ensure that data for the processor is always ready, the DMA and the processor work on separate buffers. This technique is known as double buffering which is widely used for graphics applications [27]. The processor reads and performs the necessary computations on the first buffer while the DMA is writing the next line to be processed into the second buffer (see Figure 5.5). They switch the buffers after every iteration. In this scheme, the capacity of the buffer in *vmem* is equal to two input frame lines for the warping stage.

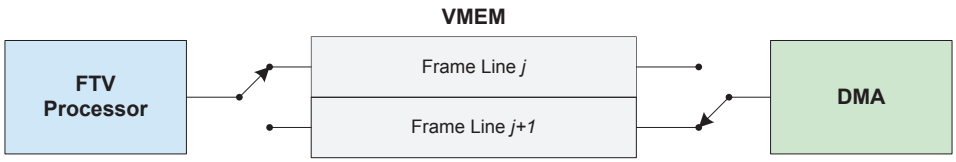


Figure 5.5: VMEM used as a double buffer between the FTV processor and the DMA

The use of vector addressable memory

Vector-addressable memory (*vamem*) is used to store $2 \times \Delta Y_{max}$ lines of the output frame. The DMA transfers one completed vamem line to the external memory during each iteration. At the same time, the processor will work on the rest of the buffer containing $2 \times \Delta Y_{max}$ output frame lines.

At the warping stage, if more than one pixel is mapped onto the same position, the one with the larger depth value should be retained. Realizing this requires a comparison between the new value and the old value residing in the vamem. Since the vamem size is limited, the same memory sections will be reused for multiple output lines. When the same section in the vamem will correspond to a new line of the frame, the values that had existed in that part should not affect the future calculations (comparisons with new depth values). To ensure this, after the completed line is transferred to the external memory, it should be reset to 0 in the vamem.

In order to realize all of the above, a circular addressing scheme is used for the vamem to avoid shifting all elements when the DMA transfers one element. In this technique, the addresses are calculated by using the modulus operation, hence the total buffer size should be a power of two to simplify the calculations. Based on the results from Figure 5.3 and the assumed circular buffering scheme, the vamem is designed to hold 64 lines of the current output frame. One vamem line is reserved for the DMA to transfer the completed line to the external memory. The rest of the buffer, 63 lines, provides a ΔY range of 31 lines, which is sufficient for this dataset.

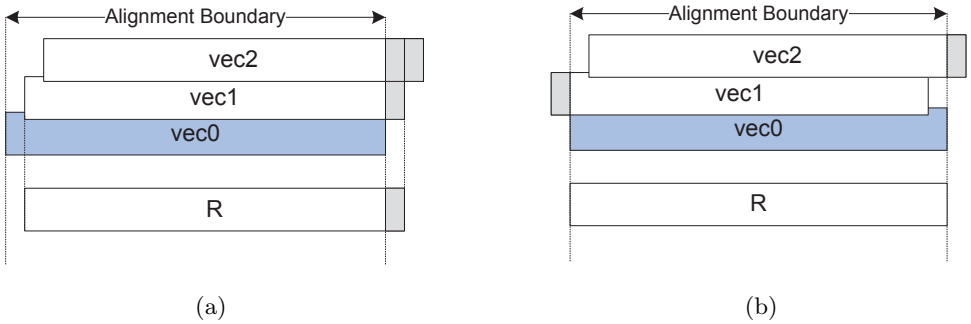
The proposed method follows these steps when processing the j^{th} line of the input frame:

- The processor sends a command to the DMA to transfer the $(j+1)^{th}$ input line from the external memory to the vector memory. The j^{th} input line was ordered during the previous iteration, so it is already available in *vmem* (double buffering).
- Each vector in the j^{th} *vmem* line is warped to the output viewpoint, i.e. their corresponding coordinates in the output frame are computed.
- Modulo-64 of these coordinates are calculated to find the actual addresses in the vamem (circular buffering).
- The values in the vector are scattered to their corresponding modulo-coordinate addresses in *vamem*.

- At the end of each iteration, the processor schedules the DMA to transfer the top buffer line $((j-32) mod 64)$ to the external memory since there can be no pixels mapped to that line any more during consecutive iterations. This vamem line is initialized back to 0 using the DMA function *init_vamem*. Even though, memory initialization takes some cycles, it is less expensive than an actual transfer between memories, and these cycles can be hidden behind the computations.

5.1.2 Unaligned Memory Accesses

Unaligned vector memory addressing is one of the critical problems in SIMD architectures [18]. It occurs when the start address of a vector load/store operation is not on the alignment boundary. Handling these accesses requires expensive memory architectures. Additional memory accesses along with special vector operations are needed to pack the desired data if such hardware support is unavailable.



(a) (b)

Figure 5.6: Modified scheme to prevent unaligned accesses

It has been shown that the vectorization of filtering applications usually results in unaligned memory accesses. In this work, 3×3 median filtering is performed which might require an unaligned access scheme. To describe how the problem occurs, consider Figure 5.6(a) which illustrates an example of 3-tap horizontal filtering. Let $vec0$ denote the vector containing pixels from the original image. The vectors $vec1$ and $vec2$ represent the right shifted version of vec0 by an amount of one and two pixels respectively. In this method, the result vector R is exceeding the alignment boundary. Therefore, storing this vector to the memory would cause an unaligned memory access. This is not supported by the current FTV architecture described previously.

In order to avoid this problem, the method shown in Figure 5.6(b) is exploited [6]. To filter the aligned vector $vec0$, the vector $vec1$ containing one element from the previous vector and the vector $vec2$ containing the first element of the next vector are constructed. The starting point of the result vector R is on the alignment boundary. Therefore, unaligned memory accesses are avoided using this scheme. Furthermore, it is required to perform only a single memory read in each iteration, since the vectors $vec1$ and $vec2$ are reused in the next iteration to obtain $vec0$ and $vec1$ respectively. The disadvantage of having this approach is the requirement of an extra slice operation. The slice operation takes two vectors to be merged and a scalar value to indicate the merging index as

operands (Figure 5.7). A negative merging index places the higher slice of the second vector to the start of the first vector.

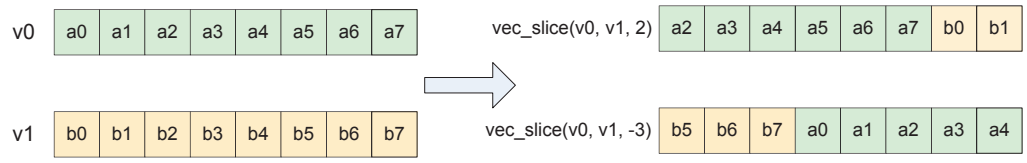


Figure 5.7: Vector slice operation

5.1.3 Boundary Behavior

In the median filtering step of the out-of-the-box algorithm, the frame boundary is extended by one pixel, filling it with 0s in order to perform filtering at the boundaries. Padding with 0s is realized by copying the array holding the image into another array. This introduces excessive memory copies.

One way to solve this problem is to handle the first and last line as well as the first and the last column of the frame separately [6], i.e. out of the main loop in the code. However this would increase the control code, therefore it degrades performance.

Another possible approach is to modify the HDMI-in interface so that it stores the incoming frame into the memory while also adding 0 values in the boundaries. This modification, however, would result in an increase in the bus-usage. Furthermore, it is an extra design effort to modify the hardware.

The more efficient and perhaps an easier method is to benefit from the use of DMA. As mentioned in the previous sections, the input to any step in the algorithm resides in the external memory. The processor schedules DMA to transfer the part of the data to the local memory. In the case of 3x3 median filtering, having three lines of the frame is sufficient to perform the filtering. This is demonstrated in Figure 5.8. While the processor performs the filtering for *input line B* and writes the result to the *output line B*, DMA handles the two memory transfers: output line A, which is the result of previous iteration, is written back to external memory, and *input line D*, the line that is needed to perform filtering of the next line, is transferred to the local memory.

The required input buffer size is $4 \times (FRAME\_WIDTH + 2 \times vector\_size)$ so that there is a place to pad additional first and last columns. Vectors of 0s are padded to the first and last vectors during the initialization of this buffer, outside of the main filtering loop. The DMA gets a line of the frame from the external memory and places it in between the first and last vectors. For padding of the first and the last line, DMA initializes the buffer line to 0 instead of transferring an actual line of the frame. The required output buffer size is 2 frame lines to realize double buffering.

The need for boundary padding is not limited to the median filtering stage of the algorithm. At disocclusion filling and dilation steps, the boundaries must be filled with non-zero values since a value of zero shows that the pixel is disoccluded. The same ap-

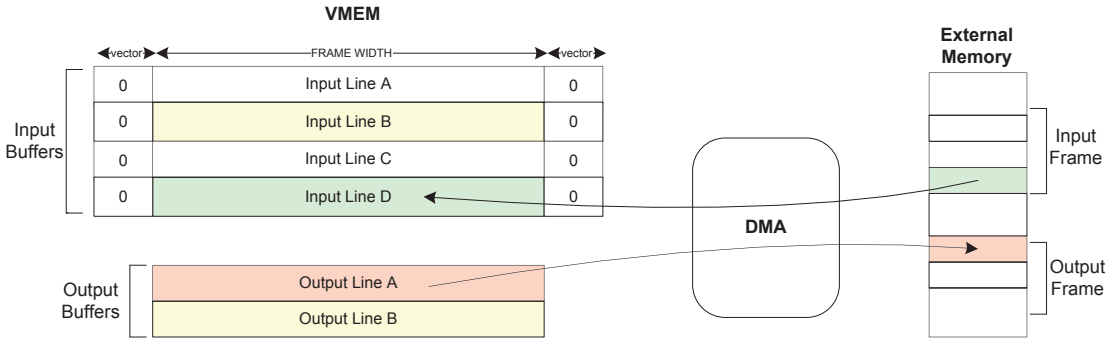


Figure 5.8: Using DMA to perform boundary padding

proach is used at the other stages of the algorithm wherever there is a need for boundary-padding.

5.1.4 Conditional Executions

The presence of conditionally executed statements is one of the reasons that degrade the level of vectorization in SIMD architectures. If-else statements inside a loop body introduce control dependencies and require particular handling for each data element. This is contradictory to the essence of vectorization which relies on the data parallelism, meaning that the same operations are performed for a number of data elements in parallel.

```
for i=1:NN
    if M[i] == 0
        M[i] = M[i] + N[i];
    end if
end for
```

Figure 5.9: An example of a conditional code

The loop shown in Figure 5.9 cannot be vectorized as its body contains conditional branches. The addition should be performed only for the elements which satisfy the condition $M[i] == 0$. Many vector processors solve this problem by converting such control dependencies into data dependencies. This is achieved with the help of what is called vector-mask operations.

Figure 5.10 depicts the transformed code from the previous example. A Boolean vector is used to hold the result of the conditional check. A 0 in any element of the flag (mask) indicates that the condition is $false$ for that vector element and 1 flag is raised when the condition is $true$. The required operations for both true and false conditions are performed. Then the final result is found by multiplexing the results of $true$ and $false$ branches using the flag. In this example, there is no else branch, therefore the elements of vector vec_M stay the same when the corresponding mask value is zero.

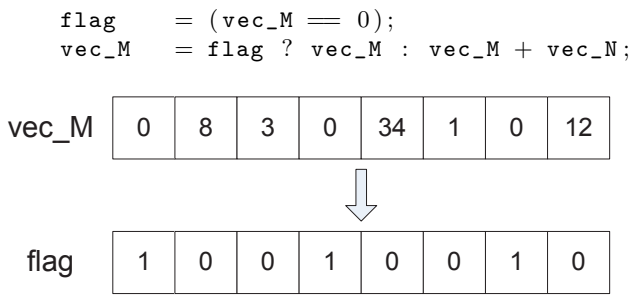


Figure 5.10: An example of vector masking

Using vector-mask operations has its advantages. It removes control dependencies in the code. This gives the compiler a larger basic block by which more instructions can be parallelized. This implies an increase in the instruction level parallelism (ILP). Furthermore, all conditionals must be removed to apply some optimizations such as software pipelining, which further increases the ILP. The drawback of having this scheme is that the vector instructions are always executed and an additional operation (multiplexing) must be performed. However, it is still faster as compared to a scalar implementation with conditionals.

In some cases, where the masking bits are rarely set to 1, the speed-up achieved by vectorization decreases due to having to execute the instructions in all control paths. In order to reduce this overhead, vector branches [30] can be inserted to bypass vector instructions, similar to how it is done in scalar branches as demonstrated in Figure 5.11. For that, intra-vector operations as shown in the example must be performed. Assuming that *vec_M* holds the pixel values ranging from 0 to 255, *vec_imin* operation returns the minimum pixel value inside the vector. If the result of *vec_imin* is larger than zero, showing that all elements of *vec_M* are non-zero, the operations inside the if-branch can be bypassed. However, it is not always possible to find an intra-vector operation that satisfies the required functionality.

```
for i=1:N
    if vec_imin(vec_M) == 0
      % If-branch Calculations
    else
      % Else-branch Calculations
    end if
end for
```

Figure 5.11: An example of a vector branch

In the FTV algorithm, a vector branching method is employed for filling disocclusions in the depth map stage. Since only a small percentage of pixels is dissoccluded, and the dissoccluded pixels are present in clusters, a vector branch check results in faster execution time than vector-masking.

5.1.5 Search Space Restrictions

The functions for disocclusion filling are not suitable for mapping onto an embedded system with limited memory due to extensive search space [6]. In order to fill a disoccluded pixel, searches are performed in horizontal, vertical and diagonal directions. The search in each direction terminates either when a non-zero depth value is found or when the search limit is exceeded. In the out-of-the-box algorithm, the search space was limited to 200 pixels in each direction. This requires a considerable amount of data stored in the local memory, and a great deal of processor cycles spent in case the search continues up to this limit.

In order to optimize the vectorized implementation for these steps of the algorithm, an analysis was performed. For the ballet stream with 1024×768 resolution, the distance of the disoccluded pixel to the pixel with non-zero value in each direction is determined for all disoccluded pixels. For the depth map, the pixel is filled from its direct neighbors in most cases. Therefore, we limited the search space to the direct neighbors of a pixel. We also modified the scheme to construct the result progressively in order to ensure that the output image quality is not reduced in the presence of clusters of dissoccluded pixels. In other words, a dissoccluded pixel which is filled in the previous iteration should be used in the inpainting of the other pixel. Assuming that the image is scanned from top-left to bottom-right, the upper neighbors and the left neighbor of the pixel is already inpainted before. Hence, there is definitely a neighbor pixel with non-zero value. An obvious drawback of this approach is favoring the top-left pixels, which results in filling pixels with the background values from top-left while in some cases it should be filled with bottom-right side. By restricting the search space with its direct neighbors during depth inpainting, the quality of the interpolated depth map is degraded (quantitative information is given in Chapter 6) while it has almost no effect on the final texture map.

Using the same method for filling the disocclusions of texture has greater effect on the output image quality as the texture values are not always filled by their direct neighbors. Another difficulty at this step is that the resulting texture value is the weighted average of the non-zero texture values, which are found in all directions using depth information, based on their distance. As a result of not using depth information and restricting the search space to the direct neighbors, the texture map quality decreased by a large degree while the subjective quality was also altered. A blurring effect is observed at the edges of the objects. However, the subjective quality in some samples showed better results since the background is spilling onto the foreground objects at some edges in the results of out-of-the-box algorithm.

5.1.6 Loop Interchange

The loop interchange technique switches the order of the loops in a nested-loop in order to increase parallelism and/or to improve data locality [21]. In the out-of-the-box FTV algorithm, the processing order is column based, i.e. the order of the nested loops was such that the image was scanned first vertically, and then horizontally as in Figure 5.12a. This would cause a problem in the vectorized implementation since the data is structured row-based in the vector memory. Therefore, the nested-loops are switched as shown in Figure 5.12b. Shifting to the row based processing, however, caused errors at the warping

stage of the algorithm. In order to avoid these errors, [23] proposes that if more than one pixels are mapped to the same output coordinate, the pixel with the largest depth value should be preserved. This solved the problem in the output depth map with the cost of an additional load and compare operation which is required to store the depth value only if it is larger than the previously written value.

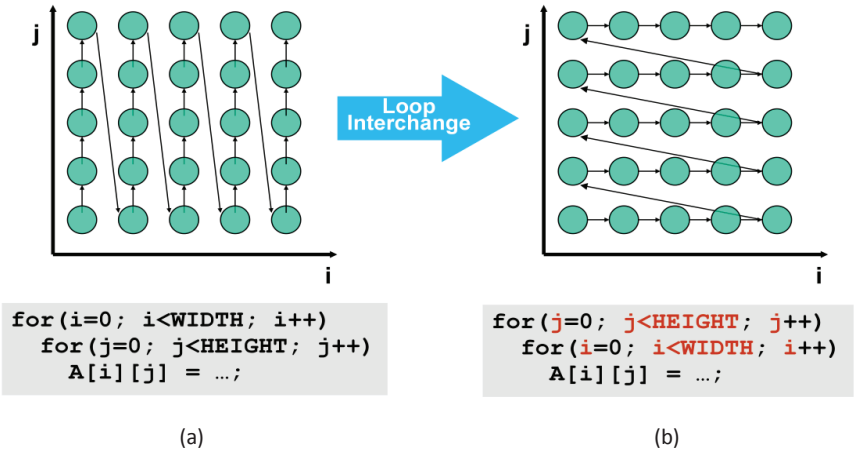


Figure 5.12: Loop Interchange

5.1.7 Performance of Vectorized Baseline

As a result of the techniques explained in the previous subsections, we achieved a full vectorization in all functions of the algorithm. We regard this implementation as our vectorized baseline, and apply several optimizations on it to improve the performance further. Performance results of the vectorized baseline for the sample ballet scene are presented in Table 5.2. Some functions of the out-of-the-box algorithm, dilate-blend and fill Y-UV, are merged during the baseline implementation. The required number of cycles per pixel is reduced to $63,64$ which corresponds to a speedup of 1.5x over the work presented in [6]. The results indicate that warping functions are still identified as the bottleneck although the maximum possible data level parallelism is achieved in these functions. The main reason is that high-latency vamem load/store operations are not yet hidden behind the computations due to the existing dependencies. For the same reason, the ILP is very low (1.67) indicating that processor resources are not efficiently utilized. The optimizations presented in the next sections will help the removal of these limitations.

5.2 Further Optimizations

5.2.1 Loop Transformations

In many multimedia or scientific applications, most of the processor cycles are spent on executing loops. Much effort is spent by compilers as well as programmers to apply a

Table 5.2: Performance of our vector baseline

	Operation Count		Cycle Count		ILP	Code Size
	Total	per pixel	Total	per pixel		
Warp	27.840.348	35,40	20.064.271	25,51	1,39	528
Blend	603.540	0,77	420.192	0,53	1,44	53
Median	3.000.864	3,82	1.505.328	1,91	1,99	118
Fill Depth	2.999.316	3,81	1.455.399	1,85	2,06	107
Inverse warp	38.950.556	49,53	23.330.026	29,67	1,67	703
Dilate + Blend	7.117.440	9,05	2.290.304	2,91	3,11	361
Fill YUV	3.168.256	4,03	980.803	1,25	3,23	147
TOTAL	**83.680.320**	**106,41**	**50.046.323**	**63,64**	**1,67**	**2.017**

group of loop manipulations, called loop transformations to optimize these critical parts. These manipulation techniques, known as loop transformations, must usually preserve the dependencies between operations in the loop body while improving the performance by:

- Reducing the loop overhead
- Increasing the number of statements that can be executed in parallel.
- Improving cache performance by enhancing data locality.

Many techniques have been developed for this purpose out of which some are analyzed in this section, and employed in order to improve the performance of our baseline implementation.

5.2.1.1 Loop-invariant Code Motion

This is a well-known compiler loop optimization that is manually applied on some parts of the code in this thesis. The idea is that a statement inside a loop body can be moved out of the loop if its corresponding reaching definition is out of the loop. A simple example code is given below. The statement 1 is independent from the inner loop while a part of the statement 2, $c2 * c3$, is a constant that is not updated either in the inner or the outer loop. Therefore, they can be moved out to the appropriate places. This results in a decrease in the number of operations and the processor cycles. This technique is applied on warping and inverse warping functions of the algorithm. Table 5.3 indicates that the cycle count is decreased by applying this technique on the code. Since the inner loop operations can be better parallelized, moving some code sections out of the inner loop reduced the ILP.

It should be noted that the operation count and the code size are not always correlated in VLIW architectures. The inner loop operations have a relatively higher effect on the operation count than the outer loop operations. Their effect on the code size, however, is the same. Hence, in our case, this technique slightly decreased the operation count while the code size is increased.

```
for (j=0; FRAME_HEIGHT; j++)
{
    for (i=0; VECTORS_PER_LINE; i++)
    {
1:      a = M[j] * c1;
2:      b = a + c2*c3;
    }
}
```

Table 5.3: The effect of invariant code motion on performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + loop invariant code motion	102,83	62,64	1,64	2.045
Improvement	**3,36%**	**1,57%**	**-1,81%**	**-1,39%**

5.2.1.2 Loop Fusion

The principle of loop fusion is based on combining the bodies of two adjacent loops when both the loops iterate the same number of times. In order to merge the loops, however, the data dependency between the statements must be preserved. A simple example is given below to demonstrate this issue. The loops in the example have a certain data dependency but their bodies can be merged without flawing these dependencies. Merging the bodies of two loops reduces the loop overhead. It also allows the compiler to achieve a schedule with higher parallelism by increasing the number of instructions that can be parallelized in the basic block.

before loop fusion	**after loop fusion**

`for i =1:N`
`  A[i] = B[i] * c1;`
`end for`
`for i =1:N`
`  C[i] = A[i-1] * c2;`
`end for`
	`for i =1:N`
	`  A[i] = B[i] * c1;`
	`  C[i] = A[i-1] * c2;`
	`end for`

The results indicate that loop fusion enhances the performance by 3,68% and improves the ILP by 3,67%. The achieved improvement in the code size is 5,16%.

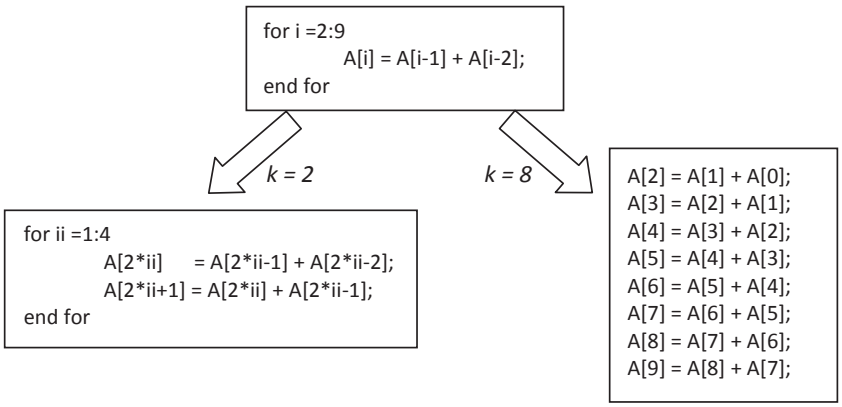
Furthermore, merging the loop bodies of two consecutive stages of the algorithm removes the need for DMA transfers when the second stage uses the output of the first stage. This is the case for merging of blend-median and dilate-blend functions. This decreases the number of processor stall cycles, which stem from the wait for loading data, in these functions. It also helps in reducing the memory bus traffic since the intermediate results are not written to the external memory.

Table 5.4: The effect of loop fusion on performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + loop fusion	106,25	61,29	1,73	1.913
Improvement	**0,15%**	**3,68%**	**3,67%**	**5,16%**

5.2.1.3 Loop Unrolling

Loop unrolling duplicates the body of the loop multiple times determined by the loop unrolling factor; k. Unrolling a loop decreases the number of times that the loop condition check is executed. It also reduces the total number of operations by eliminating a number of increments/decrements between adjacent blocks. Furthermore, as in the case of loop merging, the basic block size is enlarged; this can be used by the compiler to increase the parallelism during instruction scheduling. An example loop which is iterating eight times is given below. The code shown by the left and right arrows indicates unrolling of the loop with two different factors ($k = 2$ and $k = 8$ respectively).



The Silicon Hive compiler supports automatic loop unrolling which can be enabled/disabled by the pragma directives given below. It can be enabled only for the loops with a known number of iterations, and the unrolling factor should be an exact divisor of the iteration number.

#pragma hivecc unroll=2 : loop unrolled 2 times
#pragma hivecc unroll=off : loop unrolling disabled (by default)
#pragma hivecc unroll=full : full loop unrolling

Full loop unrolling is applied in the third level loops of warping and inverse warping stages. Besides that, loop unrolling by an integer factor is implemented for the second level loops in some functions of the algorithm. The unrolling factors of each function which give the minimum execution time while fitting into program memory and register

files are determined via experiments. Table 5.5 gives an insight about the effectiveness of loop unrolling. The cycle count is reduced by 34,24% by loop unrolling mainly because the compiler is enabled to schedule more operations in parallel with the increased basic block size. A larger improvement in the operation count caused a 3,16% decrease in the ILP. Due to duplicating the loop bodies, the code size increased by 32,72%.

Table 5.5: The effect of loop unrolling on the performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + unrolling	67,76	41,85	1,62	2.677
Improvement	**36,32%**	**34,24%**	**-3,16%**	**-32,72%**

5.2.1.4 Loop Folding (Sofware Pipelining)

This is another efficient technique to improve instruction level parallelism. In software pipelining, the operations of each iteration are split into *s* stages, and stage 1 from iteration *i*, stage 2 from iteration *i-1*, etc. is performed in a single iteration [5]. This would imply a parallelism in between consecutive iterations of the loop in addition to the parallelism within the loop body which is the case in the absence of software pipelining.

In order to illustrate the effectiveness of this technique, let us consider the following loop in C-like assembly code [9].

```
L1: a = *p++
    b = a + 1;
    *q++ = b;
    if(--n != 0) goto L1
```

The parallelism in this loop body is very limited. Scheduling the loop without software pipelining might result in the following schedule. Considering this in a VLIW machine, the operations in the same line belongs to one large instruction which is executed in a single cycle. The schedule length is 3 cycles for every execution of this loop body.

```
L1: a = *p++
    b = a + 1;
    *q++ = b; if(--n != 0) goto L1
```

The figure below depicts the scheduling of this loop for ten loop iterations with software pipelining enabled assuming that the VLIW processor has sufficient hardware resources to execute the operations in parallel.

The code above is for illustration purposes. Since it is rather large in code size, the compiler generates the following code instead.

This example can be used to analyze the advantage of software pipelining. In the initial scheduling, 4 operations are performed in 3 cycles, giving an ILP of 4/3. A schedule with software pipelining for ten iterations, performs 40 operations in 12 cycles,

```
a = *p++
a = *p++; b = a + 1;
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
a = *p++; b = a + 1; *q++ = b; if(--n != 0) .
          b = a + 1; *q++ = b
                     *q++ = b
```

```
    n = n - 2;
    a = *p++
    a = *p++; b = a + 1;
L1: a = *p++; b = a + 1; *q++ = b; if(--n != 0) goto L1
              b = a + 1; *q++ = b
                         *q++ = b
```

the ILP is 40/12 in this case. As the number of iterations increases, the ILP converges to 4 since the loop effect of the prologue and epilogue reduces. Therefore it improves the parallelism by a factor of 3 in this example.

The Silicon Hive compiler automates enabling software pipelining with the following pragma which needs to be included at the end of the loop to be pipelined. In order to enable this option, there should be no control statements, i.e. if-else statements, loops, function calls, in the loop body.

#pragma hivecc pipelining=0

In some functions of the algorithm, as a result of removing conditionals and fully unrolling the third level loops, the loop body does not have any control flow. Moreover, all iterations of the loop are independent, i.e. the computations in any given iteration do not depend on the results of the previous iteration. This makes those loops perfect candidates for software pipelining. Table 5.6 shows how much improvement was achieved by this technique over our baseline. Software pipelining resulted in an increase in the code size due to the additional codes included before and after the pipelined loop. Nevertheless, it accelerates the code by 3,09% along with a 1,22% improvement in the ILP.

Table 5.6: The effect of software pipelining on the performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + SW pipelining	104,38	61,67	1,69	2.166
Improvement	**1,91%**	**3,09%**	**1,22%**	**-7,39%**

5.2.1.5 Loop Retiming

As presented previously, loop unrolling and software pipelining techniques improve the performance of loop executions by means of increased parallelization between iterations. However, the resulting schedule is still not optimal in warping functions; i.e. there are still processor stall cycles in the schedule. Even with a considerable amount of operations in the loop body, high latency VAMEM load/store operations are not totally hidden. The compiler could not find a way to perfectly pipeline consecutive iterations with the code given. For this reason, the loop retiming technique is employed manually to improve the performance.

Loop retiming, first proposed by [28], is used to regroup the order of execution in a loop body in such a way that parallelism is increased while preserving the dependencies between operations. A simple example is given in Figure 5.13 to demonstrate how it makes several parallelism opportunities possible.

Before loop retiming (Figure 5.13a), statement 2 must wait until statement 1 is executed, likewise statement 3 and 4 must be executed after statement 2. The corresponding multi-dimensional data flow graph (MDFG) which models the dependence between operations is also given. In an *MDFG*, each node represents an operation, and edges indicate the dependency between nodes. An edge e from u to v with weight $d(e) = (d_i, d_j)$ means that the computation of node v at iteration *(i,j)* depends on the result of node u calculated at iteration $(i - d_i, j - d_j)$. Achieving full parallelism in the loop body is equivalent to obtaining non-zero *d(e)* for any $e \in E$ where E denotes the set of dependence edges [28]. Therefore, only statement 3 and statement 4 can be executed in parallel in the first case.

Applying a chain of retiming resulted in the code and the corresponding *MDFG* shown in Figure 5.13b. The MDFG indicates that all edges have a non-zero value in terms of the inner loop index j. Therefore, all four statements in this loop body can be executed in parallel. In order to avoid a loss or gain of some statements during the transformation, the second level loop index is rearranged, and short pieces of code are inserted outside the nested loop. These code pieces placed before and after the second level loop are called *prologue* and *epilogue* respectively.

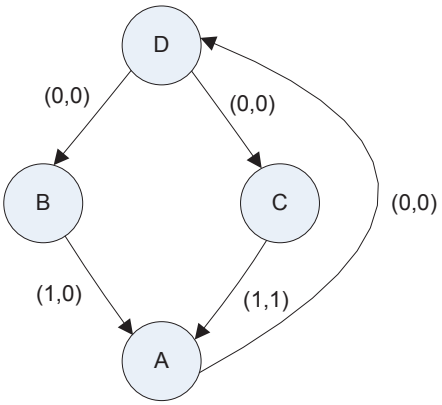
Table 5.7 shows the effects of loop retiming on performance and code size. Overall, the retiming improved the performance by 10,92%. The code size is increased by 29,60% due to the additional prelogue and epilogue segments required after the retiming of the loop body. As it will be presented in the next sections, applying retiming along with other optimizations will have an even higher effect on the performance.

Table 5.7: The effect of loop retiming on performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + retiming	103,38	56,69	1,82	2.614
Improvement	**2,85%**	**10,92%**	**9,06%**	**-29,60%**

before retiming

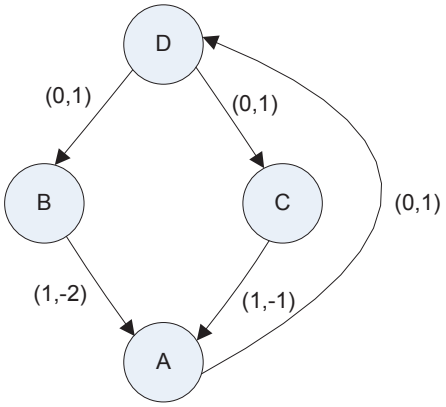
```
for i=1:N
  for j=1:M
  1:  A[i,j] = B[i-1, j] + C[i-1, j-1];
  2:  D[i,j] = A[i,j] * c1;
  3:  B[i,j] = D[i,j] + c2;
  4:  C[i,j] = D[i,j] + c3;
  end for
end for
```



(a)

after retiming by r(A)=2, r(D)=1

```
for i=1:N
  % Prelogue
  A[i,1] = B[i-1,1] + C[i-1,0];
  A[i,2] = B[i-1,2] + C[i-1,1];
  D[i,1] = A[i,1] *c1;
  for j=1:M-2
  1:  A[i,j+2] = B[i-1,j+2] + C[i-1,j+1];
  2:  D[i,j+1] = A[i,j+1] * c1;
  3:  B[i,j]   = D[i,j] + c2;
  4:  C[i,j]   = D[i,j] + c3;
  end for
  % Epilogue
  B[i,M-1] = D[i,M-1] + c2;
  B[i,M-1] = D[i,M-1] + c2;
  D[i,M]   = A[i,M] * c1;
  B[i,M]   = D[i,M] + c2;
  B[i,M]   = D[i,M] + c2;
end for
```



(b)

Figure 5.13: An example code and the corresponding MDFG indicating loop retiming

5.2.2 Data Mapping

In the FTV processor, each of the data memories is accessible by only one of the load-store units. If all the variables are defined in the same memory, then it is not possible to access these variables at the same time. This is especially important for accesses to vector addressable memory, *vamem*, which has a latency of 18 cycles in a 16-way architecture. In our baseline vectorized implementation, vamem data were defined in two separate *vamems*: the first large vamem holding the scattered data during warping stage and the second reserved for a small look-up table (LUT). The inverse-warping stage of the algorithm requires six accesses to this memory as depicted in the code below. Scheduling this loop body would result in at least 4×18 cycles in the baseline configuration since 4 vamem accesses (depth, y, u, v) were required to share the same memory. Distributing the critical data elements over multiple memories can help reducing this bound in the latency.

An important consideration while distributing data is that having multiple data memories would imply a larger area as compared to a single memory with the same size of

the sum of multiple memories. Introducing additional memories would also require additional load/store units associated with them. Therefore, a trade-off exists between area and performance while introducing multiple memories to achieve better data mapping.

In this specific case, in order to prevent *vamem* accesses from becoming a bottleneck, the data is distributed over three different memories, *vamem1*, *vamem2* and *vamem3* as shown in the example code. It is possible to specify for each data the memory that it will be placed in, using the attribute *MEM(MEM_NAME)*. The groups of two variables share the same *vamem*. In this way, a schedule with 2×18 cycles latency can be achieved, assuming that other operations in the body can be executed in parallel to these load/stores. More than three memories would not be needed since the other computations in the loop body cannot be executed in less than 2×18 cycles with the current processor configuration. Therefore, using three memories is an optimum choice regarding area and performance.

```
unsigned char MEM(VAMEM1) depth_buffer;
unsigned char MEM(VAMEM1) y_buffer;
unsigned char MEM(VAMEM2) u_buffer;
unsigned char MEM(VAMEM2) v_buffer;
unsigned char MEM(VAMEM3) LUT_depth;
unsigned char MEM(VAMEM3) LUT_division;

for (j=0; FRAME_HEIGHT; j++)
{
    for (i=0; VECTORS_PER_LINE; i++)
    {
        a = VAMEM_LOAD(LUT_depth[i]);
        b = f(a); % A number of computations
        c = VAMEM_LOAD(LUT_depth[b]);
        d = f(c); % A number of computations

        depth = VAMEM_LOAD(depth_buffer[d]);
        y     = VAMEM_LOAD(y_buffer[d]);
        u     = VAMEM_LOAD(u_buffer[d]);
        v     = VAMEM_LOAD(v_buffer[d]);

        result= f(depth,y,u,v);
    }
}
```

The data mapping method is used in the inverse warping stage only due to a large number of vamem load/store operations present in this function. The individual contribution of data mapping on the performance is 6,09% as shown in Table 5.8.

5.2.3 Extending the ISA

A common technique to enhance the execution time of an algorithm is to extend the processors' instruction set architecture (ISA) with custom instructions. Custom instructions are usually specific to an application-domain, and they are the combination of some simple operations. They have some advantages such as reducing the code size and the

Table 5.8: The effect of data mapping on the performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + data mapping	105,34	59,76	1,76	1.959
Improvement	**1,00%**	**6,09%**	**5,42%**	**2,88%**

number of instructions being issued. Furthermore, the register usage can be reduced since the need to hold the intermediate variables is avoided. Using custom instructions, however, enlarges the logic area of the processor as it usually requires additional functional units. This can be an important overhead if the same processor will be used in other application domains. However, it is not the concern of this thesis since its focus is on the efficient implementation of the FTV algorithm. Furthermore, the target platform is an FPGA; hence extending the ISA is acceptable as long as it fits on the FPGA while satisfying the timing constraints.

According to Amdahl's law [2], in order to improve the performance of a code, the effort should be spent on the segment which is executed the most frequently. Considering this fact and the saving in terms of processor cycles in each execution of the custom instruction as well as its feasibility, a single custom operation is proposed in this work.

5.2.3.1 Vector Leading One Detection (OP_vec_lod)

The Newton-Raphson division approximation present in the fixed point implementation requires finding the leading one position, i.e. the position of the most significant digit with the value 1, of the divisor. In the absence of a dedicated instruction for this operation, a possible way is to perform shift operations x times, where x equals to number of bits in a vector element. The code segment below demonstrates this method. The variable *vec_lod* is the vector holding the leading one position for each element of *vec_x*. A flag and mux operation is required to detect when the element value reaches zero, and *vec_lod* is incremented for the elements which does not satisfy this condition.

```
vec_lod = 0;
for i=1:bits_per_vecelem
   flag_lod = (vec_x == 0);
   vec_lod  =  flag_lod ? vec_lod : vec_lod+1;  % Increment the counter
        if it is still nonzero
   vec_x     = vec_x >> 1;
end for
```

The execution of the code segment described above requires $5 \times bits\_per\_vecelem$ operations which takes a considerable amount of cycles even if the loop is unrolled. Furthermore, the frequency of this operation in the algorithm is four per pixel. Having a dedicated operation, *OP_vec_lod*, can improve the performance significantly. Also, the operation can be easily implemented in hardware using a priority encoder.

In order to run software simulations of the FTV algorithm with the new custom instruction, the semantic description shown in Figure 5.14 was added into the processor description file (TIM). Using this semantic, both functionality and the cycle property of the operation are known to the compiler and simulator. The next step in the flow of introducing a new operation is to write the RTL description of the operation to generate the hardware block. However, due to lack of time remaining in the project, the hardware unit for it was not implemented.

```
OP vec_lod <unsigned nway> (svecN<nway> A) -> (uvecN<nway> R)
{
    SEM R<nway> (A) =
    {
       unsigned <Width:=18> t;
       for i [(nway-1),0]
       {
            t=0;
            for j [17, 0] { if(Bits(A[i],[17-j])) then { t = 17-(
                unsigned<Width:=18>)j; } fi }
            R[i]=t;
        }
    };
};
```

Figure 5.14: The semantic description for vector leading one detection operation

Table 5.9 indicates the achieved improvement over our baseline by introducing this operation to the processor ISA. The total cycle count is reduced by 29,12%. The decrease in the operation count is even higher, which caused a lower ILP. The code size is also decreased since the same functionality is achieved by a single line of code instead of the expensive loop mentioned above.

Table 5.9: The effect of OP_vec_lod operation on performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
baseline	106,41	63,64	1,67	2.017
baseline + OP_vec_lod	59,57	45,11	1,32	2.008
Improvement	**44,02%**	**29,12%**	**-21,02%**	**0,45%**

5.2.4 The Combined Effect of Optimizations

A combination of the optimization methods mentioned above is applied on the vectorized baseline.

For the warping and inverse warping stages, we first used the loop invariant code motion technique to reduce the amount of inner loop operations. Next, the custom operation OP_vec_lod is also introduced to additionally reduce the inner loop operations.

These optimizations allowed us to apply loop unrolling and loop merging techniques without introducing data movements between the register file and memory or introducing unacceptable memory requirements. These two techniques were applied when possible in order to increase the basic block sizes and thereby enable additional parallelism between operations. The schedule generated by the compiler contained huge amount of wait cycles due to latency induced by vamem load/stores (the schedule is listed in Appendix 7.2). By retiming the loop, these latencies can be completely hidden and an almost optimal schedule can be achieved for the warping stage as shown in Appendix 7.2. The inverse warping stage, however, was still not optimal due to the additional vamem accesses. These accesses are required to load depth, Y, U, and V pixels. An efficient data mapping technique helped us to remove this limitation by distributing the data requests over two vamem structures of equal size keeping the overall vamem space the same.

Furthermore, the classical techniques of loop unrolling, software pipelining, or a combination of the two was applied on the remaining algorithm stages. The selection of which technique to apply was made by analyzing the corresponding compiler-generated schedules. These stages were, however, not our primary target.

Table 5.10 shows the overall improvement achieved over our baseline implementation. The details for the performance of each algorithm function are given in Chapter 6. Overall, the cycle count and the ILP are improved by 75%. Some of the optimizations, particularly loop unrolling and loop retiming, require a trade-off of code size to improve the performance, hence the total code size is 23,65% larger than our baseline in the final implementation.

Table 5.10: The combined effect of optimizations on performance and code size

	Operation Count per pixel	Cycle Count per pixel	ILP	Code Size
Baseline	106,41	63,64	1,67	2.017
Baseline + combined optimizations	46,34	15,76	2,94	2.494
Improvement	**56,45%**	**75,23%**	**75,82%**	**-23,65%**

6 RESULTS & ANALYSIS

This chapter presents the results of mapping the FTV algorithm on the FTV processor. It starts with giving the performance results for each stage of the algorithm. Next, a performance comparison of this work with the previous vectorized work and the out-of-the-box code is given. This is followed by area results indicating the FPGA resource utilization. Finally, the final output image quality of the vectorized algorithm is compared to that of a scalar fixed point implementation taking the floating point implementation as a reference. We discuss the reasons for output degradation in certain stages.

6.1 Performance Results

After vectorizing the FTV algorithm and improving its performance through the number of optimizations described in Chapter 5, its performance is measured via a *sched* simulation. For the sample ballet stream with a $1024x768$ resolution, Table 6.1 gives the operation count, cycle count, ILP and code size for each function of the algorithm. Since the FTV processor has scalar and vector functional units, the total operation count for both is indicated separately. The total number of cycles, that the algorithm takes, is the sum of execution cycles and stall cycles. The execution cycles are the cycles in which the processor is executing a number of operations and the stall cycles represent the cycles for which the processor has to wait for memory transfer latencies (in this case it is the DMA transfers between the external memory and the processor local memory). In a *sched* simulation, stall cycles are calculated based on the system-level parameters given during the instantiation step of the buses and the memories in the *hsd* description. It reflects the amount of latency due to a read from the external memory, and a bus transfer. However, this model is not sufficient to represent the real-time behavior of the system since it does not take into account the latencies in case the memory bus is being employed by another device, e.g. the HDMI interfaces.

Table 6.1: The performance of the implementation for each algorithm step

	OPERATION COUNT			CYCLE COUNT		ILP			
	Scalar	Vector	Total	Execution Cycles	Stall Cycles	Scalar ILP	Vector ILP	Total ILP	Code Size
Warp	807.936	9.632.256	10.440.192	3.937.526	12.875	0,21	2,45	2,65	781
Blend+Median	505.344	2.714.112	3.219.456	1.031.548	1.296	0,49	2,63	3,12	213
Fill Depth	583.680	1.116.672	1.700.352	780.457	480	0,75	1,43	2,18	115
Inverse warp	2.654.208	10.368.000	13.022.208	4.380.723	296.616	0,61	2,37	2,97	1153
Dilate+Blend	457.728	5.058.048	5.515.776	1.481.256	453.870	0,31	3,41	3,72	439
Fill YUV	495.360	2.048.256	2.543.616	784.521	407.296	0,63	2,61	3,24	175
TOTAL	**5.504.256**	**30.937.344**	**36.441.600**	**12.396.031**	**1.172.433**	**0,44**	**2,50**	**2,94**	**2876**

The ILP per function is also given in the table, indicating the instruction-level parallelism in scalar and vector operations separately. The vector ILP is more important

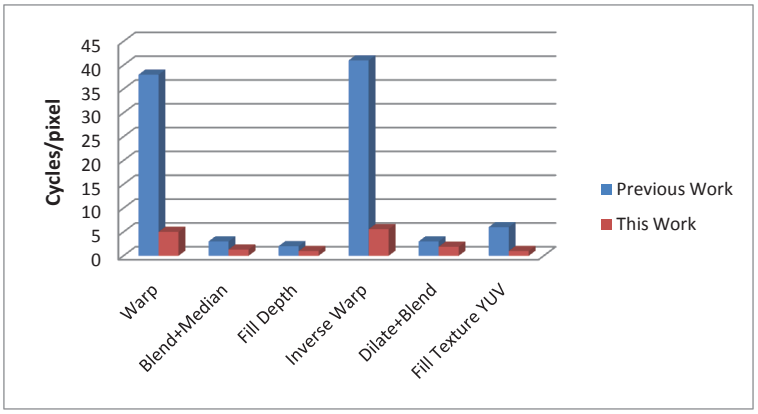


Figure 6.1: A performance comparison between this work and the previous vectorized implementation

since most of the operations are performed on vectors. The overall vector ILP of 2.5 is achieved although the FTV processor has 5 vector issue slots. Profiling the FTV algorithm indicates that the most-utilized functional units are *vshu* (vector shift unit), *varu* (vector arithmetic unit) and *vmul* (vector multiplication unit). The operations performed in these functional units account for around 70 percent of all operations in the algorithm. In the current issue-slot configuration, these functional units, two *varu*, one *vmul* and one *vshu*, are present in three issue slots. Therefore, the utilization of these three issue slots is quite high as compared to the other two. This is the main reason due to which the vector ILP is limited to 2.5. Introducing a relatively low-cost *varu* and *vshu* to the other issue slots would help in improving the performance further.

In terms of the code size, the warping and the inverse warping stages are the bottlenecks. This is because of the long prologue and epilogue codes introduced by two levels of loop retiming. Two levels of loop retiming are required since it is applied together with loop unrolling with a factor of two.

Comparison with the Previous Work

In this section, we compare the performance of our final design with the previous work [6] implementing the FTV algorithm on the same architectural template. As discussed previously, the reference implementation suffered from insufficient vectorization at the warping and inverse stages of the algorithm due to irregular memory accesses. With the proposed caching mechanism to handle these random accesses, those stages were fully vectorized in this work. The graph in Figure 6.1 illustrates the improvement in all functions, while knowing that the main contribution is coming from overcoming the bottleneck at the warping stages.

A more detailed comparison for each function including the operation and cycle count per pixel, as well as the ILP, is presented in Table 6.2. Since the reference work does not specify any information about the distribution of operations between the vector and scalar units, the total operation count and the overall ILP are given in the table.

Furthermore, the reference work was based on the assumption that the frames and all intermediate results were stored in the local memory. Therefore, the stall cycles caused by system-level aspects such as DMA transfers were not taken into account in the previous work. For that reason, in order to have a fair comparison, the table demonstrates only the execution cycles.

Table 6.2: Performance comparison for each algorithm function

	Previous Work [6]			This Work			SPEED-UP
	Operation/pixel	Cycle/pixel	ILP	Operation/pixel	Cycle/pixel	ILP	
Warp	86,00	38,00	2,26	13,28	5,01	2,65	**7,59**
Blend	1,00	1,00	1,00	4,09	1,31	3,12	**2,29**
Median	4,00	2,00	2,00				
Fill Depth	2,00	2,00	1,00	2,16	0,99	2,18	**2,02**
Inverse warp	118,00	41,00	2,88	16,56	5,57	2,97	**7,36**
Dilate	7,00	3	3,51	7,01	1,88	3,51	**1,59**
Blend							
Fill Y	4,00	3	1,33	3,23	1,00	3,24	**6,01**
Fill UV	5,00	3	1,67				
TOTAL	**227,00**	**93,00**	**2,44**	**46,34**	**15,76**	**2,94**	**5,90**

The performance of the critical stages, namely the warping and the inverse warping is improved by a factor of 7.59 and 7.36 respectively. The key for this speed-up is full vectorization by handling irregular memory accesses. Together with that, software optimizations and introduction of custom operations allow for efficient utilization of the processor's resources and hiding vamem load/store latencies, thereby increasing the ILP. Merging the blending and median filtering stages, along with a more efficient DMA-based boundary padding scheme, yields a speed of 2.29. The new boundary padding scheme also improved the performance in the other steps: filling disocclusions for depth and YUV, and the dilate-blend step. Overall, a speed-up of 5.90 is achieved and the ILP is improved by 20 percent.

Table 6.3 summarizes the results for the out-of-the-box implementation, reference vectorized implementation [6], and this work. The out-of-the-box algorithm, running on a 5-issue slot floating-point processor, reaches up to a frame rate of 0.10 fps for a $1280x720p$ resolution, while the previous vector implementation can go up to 1.17 fps. In this work, the effective frame rate is equal to 6.89. This would imply that 12.2 percent of a $720p$ stream can be covered if the rendering is performed at 30 fps per eye.

6.2 Area results

The FTV system is mapped onto the Xilinx Virtex-5 FPGA platform. The logic blocks of the processor are translated to the slice LUTs and the slice registers while the multiplication operations are performed by the DSP48E slices, each of which contain a 25×18 multiplier. 18k and 36k Block RAMs on this FPGA are used as processor local memories.

After the placement and routing, the maximum frequency is 50.078 MHz, given a 50 MHz clock constraint. The resource utilization of the FPGA is given in Table 6.4. The results prove that the current system is able to fit on the target FPGA. In terms of the

Table 6.3: Overall performance comparison

	Out-of-the-box	**Previous Work**	**This Work**
Operations/pixel	1410	227	46
Cycles/pixel	1049	93	16
ILP	1,35	2,44	2,94
Pixels/second	41,667	537,63	3,172,588
720p frame rate (fps)	0,09	1,17	6,89
Screen Coverage for 720p @ 30 Hz per eye	0.17%	1.9%	11,5%

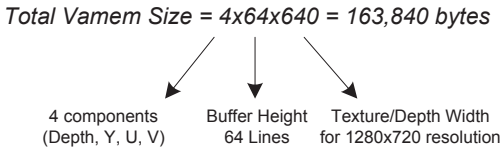
utilization of logic slices, there is still some space which might allow for the extension of the processor resources. Increasing the number of ways (vector length) in the SIMD datapath, or introducing new issue slots might increase the performance of the algorithm further.

Table 6.4: FPGA resource utilization

Slice Logic Utilization:	
Number of Slice Registers:	31,709 out of 207,360 15%
Number of Slice LUTs:	77,568 out of 207,360 37%
Number of DSP48Es:	35 out of 192 18%
IO Utilization:	
Number of bonded IOBs:	338 out of 1,200 28%
Specific Feature Utilization:	
Number of BlockRAM/FIFO:	128 out of 288 44%
Number of 36k BlockRAM used:	90
Number of 18k BlockRAM used:	39
Number of 36k FIFO used:	18
Total Memory used (Kbits):	4590 out of 10,368 44%

The area results also indicate that 44 percent of the FPGA block rams are occupied by local memories. This is based on the memory sizes given in Table 6.5. The table also shows the memory utilization based on the simulation results for the program memory (*PMEM*), the scalar data memory (*DMEM*), the vector memory (*VMEM*) and the vector-addressable memories (*VAMEM*). With the current issue slot and register file configuration, the word length (WL) of the *PMEM* is 355 bits. The scalar data path and therefore the *DMEM* is 32 bits wide. The WL of the vector memory is 288 bits, sufficient to store a single vector, which consists of 16 elements each of which has an 18-bit element precision. Finally, there are three vector-addressable memories. The first two are used as data-caches storing the pixel values; therefore they are 8 bits wide. The third *VAMEM* is used as a look-up table for the algorithm parameters (depth conversion table) and for the Newton-Raphson division approximation coefficients.

The experiments explained in Chapter 5 indicated that, for this data-set, the sizes of data vector-addressable memories (*VAMEM1*, *VAMEM2*) can be chosen such that they can store 64 lines of the output frame consisting of the depth, Y, U and V components. Therefore, the total required *VAMEM* size is given by Equation 6.1. This number is equivalent to 42 percent of the total memory size used. It should be noted that, in order to speed-up the inverse warping stage (as presented in Section 5.2.2), data is distributed over two equal sized memories. *VAMEM1* holds the depth and the Y pixels while *VAMEM2* stores the U and the V values.



(6.1)

Table 6.5: Processor local memory sizes and their utilization for the ballet sample

MEMORY	**Capacity (Bytes)**	**Utilization (Bytes)**	**Utilization (%)**
PMEM (355 bit)	181.760	139.116	76,53
DMEM (32 bit)	16.384	3.649	22,27
VMEM (288 bit)	131.072	78.240	59,69
VAMEM1 (8 bit)	81.920	81.920	100,00
VAMEM2 (8 bit)	81.920	81.920	100,00
VAMEM3 (18 bit)	4.096	1.616	39,45
TOTAL	**497.152**	**386.461**	**77,73**

6.3 Output Quality

Converting the original floating-point algorithm into fixed-point introduces intermediate rounding errors, which reduces the image quality. The quality is further decreased after mapping the fixed-point algorithm onto the vector architecture. The reasons for the errors introduced by vectorization are explained further in this section. Other than subjective ways, the degradation in output image quality can be calculated using two widely-used metrics. PSNR (peak signal-to-noise ratio) is the first method specified by the following equation:

$$\begin{aligned} PSNR &= 20\ log\frac{255}{\sqrt{MSE}} \\ \text{with } MSE &= \frac{1}{N}\sum_{x,y\in S}(Y_v(x,y) - Y_{ref}(x,y))^2 \end{aligned} \tag{6.2}$$

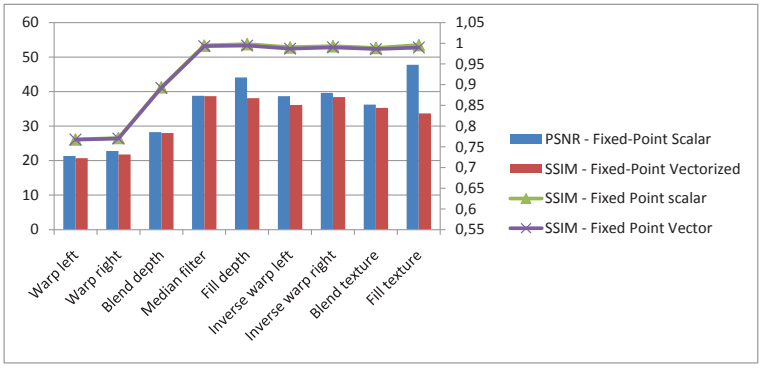


Figure 6.2: PSNR and SSIM measurements after each algorithm step

The parameter S denotes the image space, N represents the number of pixels within S, Y_v and Y_{ref} are the luminance of the reconstructed and the reference image, respectively. PSNR is the most widely used metric to detect objective quality. However, this evaluation model calculates the quantization error, while the impact of subjective feelings are never taken into consideration [33].

The SSIM (Structural Similarity) index is another technique for measuring the level of similarity between two images. It is a more complicated and precise technique as compared to PSNR, designed to take the human visual system into account. The SSIM is calculated on various windows of an image, and the resultant index is between -1 and 1, where the value 1 represents identical images.

6.3.1 Output Quality after Each Algorithm Step

In order to evaluate the output image quality after each step of the algorithm, the results of the floating-point algorithm are used as references. By averaging the results of the experiment repeated on a 100 frames, the plot in Figure 6.2 is obtained. The blue bar indicates the output PSNR as a result of the fixed-point conversion. The red bar represents the change in PSNR after the vectorization and optimizations explained in Chapter 5. The lines represent the SSIM indices for respective implementations. In the vectorized algorithm, the steps that affect the output image quality are warping, filling disocclusions in the depth map, and filling disocclusions in the texture map.

The reason for the quality difference at the warping stage is that vectorizing the load/store operations might actually alter the behavior of the algorithm. At this stage, if more than one pixel is mapped onto the same position, the larger depth value should be retained. In order to ensure this, first a load from the calculated address is performed. The loaded value is compared with the new value, and the new value is stored only if it is larger than the old value. In a scalar load/store pattern, every pixel can be compared with the most up-to-date value, hence the value with the largest depth can always be preserved. After vectorizing the load/store, however, if two elements in the same vector are mapped to the same output pixel, then the last written element is retained. Figure 6.3 shows an example of this phenomenon. In the initial condition, the vector addressable

memory already contains a number of values, and there is a pending request to store the new values residing in *vec_data* into the address specified by *base* and *vec_index*. First, a load operation (*OP_vmldo*) is performed. The element-wise maximum of the *loaded vector* and *vec_data* is chosen using *OP_vec_max* operation. The resultant *vec_max* vector is written back to the memory consecutively, i.e. one element is written in every cycle. By this method, the values in the latter elements always overwrite the former values in case they will be written to the same memory location, as highlighted with red arrows and rectangles in the figure. Therefore, the larger values are not always retained. This hazard occurs only if more than one pixel inside the same vector is mapped to the same position, and it reduces the output quality by around 1 dB. However, after the blending and the median filtering, the effects tend to disappear in the final interpolated depth map.

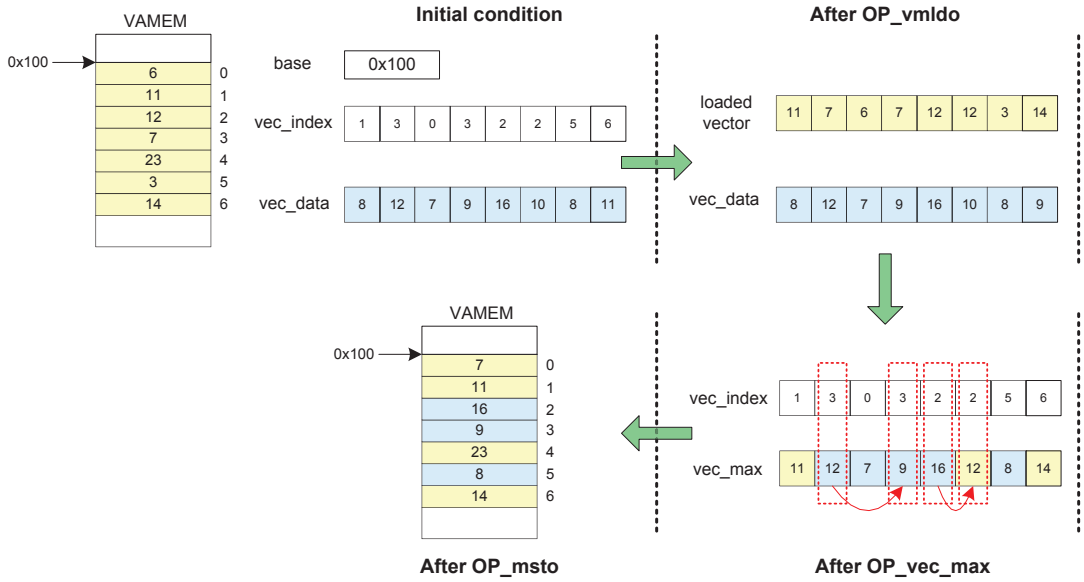


Figure 6.3: VAMEM hazard affecting the result when two elements of a vector are mapped to the same memory location

Restricting the search space to the immediate neighbors degrades the output quality of the function to fill disocclusions in depth map. The difference in PSNR is 6 dB at this step. This error also propagates through the inverse warping stage which uses this interpolated depth map. However, after the blending of the textures, the error is still less than 1dB.

The main effect on the final image quality stems from the step where the disocclusions in the texture map are inpainted. As discussed in Chapter 5, the search is restricted to direct neighbors at this step in order to reduce the extensive computational requirement. This results in a blurring effect at the edges of the foreground object. The objective quality in terms of PSNR decreases by 14 dB. The SSIM index goes down from 0.995 to 0.989. The subjective quality is also altered; however, in some cases the output subjective quality is better after vectorized implementation.

6.3.2 Final Output Quality

In order to evaluate the final output quality, the reference image is chosen to be the original camera view at the 3^{rd} camera position. Based on this reference, PSNR and SSIM indices are evaluated for the FTV algorithm with floating-point, fixed-point scalar, and fixed-point vectorized implementations. Table 6.6 indicates that the quality difference between the floating point and our fixed-point vectorized implementation is very small in terms of PSNR and SSIM.

Table 6.6: Final output quality in terms of PSNR and SSIM

	Float	**Fixed-Scalar**	**Fixed-Vector**
PSNR	33,85	33,74	33,31
SSIM	0,90374	0,90309	0,90158

7 Conclusions and Future Work

This chapter concludes the thesis by summarizing the major contributions and the achieved results. It also suggests a number of points for further improvements in the future work.

7.1 Conclusions

In this thesis, an architecture to realize an interactive free-viewpoint interpolation is developed. A Full HD specification and intensive computation of the algorithm demand a very high processing power. Furthermore, providing flexible solutions with a shorter development cycle is an important consideration in this field due to continuous algorithmic improvements and the absence of a stable free viewpoint 3D-TV standard. Our platform contains a processor from Silicon Hive image signal processor (ISP) family promising a high performance along with full programmability. It is powered by a 7-issue slot VLIW architecture combining scalar and vector data paths.

The free-viewpoint algorithm developed for the iGLANCE project was vectorized and mapped onto this processor. In the warping functions of the algorithm, pixel based load/stores requiring irregular memory accesses prevented the previous work from vectorizing these functions. This caused a significant performance drop. In order to address this issue, we used a scatter-gather unit and a customized memory transfer scheme. This, along with several classical vectorization techniques, allowed us to utilize data level parallelism completely. We regard this as our vectorized baseline. Although a certain speed-up was achieved by improved data parallelism, dependencies of the rest of the computations on high latency vector addressable memory load/stores limited the performance. In order to solve this problem and to improve instruction level parallelism, further optimizations were applied. This includes loop transformations, an efficient data mapping for critical sections, extending the processor instruction set. As a result of all these, a speed-up of a factor 6x is achieved over the selected baseline, which is equivalent to 78x over the out-of-the box code. The ILP is also improved by 17% as compared to the previous work.

We also set up a demonstration system to validate the results in a real-time environment by mapping our design to an FPGA running at 50 MHz frequency. We demonstrated that our design is occupying only 15% of the register slices and 37% of the LUTs on Xilinx Virtex-5 XC5VLX330. The achieved frame rate on this FPGA is 6.75 fps in a 1280x720 resolution. This implies that when our design is mapped to silicon, running at about 10x the FPGA frequency and with extended processor resources, we will be able to achieve performance levels required for Full HD processing at 30 fps per eye.

7.2 Future Work

We demonstrated with the area results that there is still scope to extend the processor resources by increasing the number of ways and/or additional issue slots. Profiling the FTV algorithm showed that introducing an additional vector multiplier, a vector shift unit, and a vector arithmetic unit will result in a better load distribution over the current issue slots, hence it will increase the ILP and the performance. However, this acceleration would be valid only for the computation parts. There would be still a latency bound by vamem load/stores. To improve this part too, the vector addressable memory can be split into multiple banks or the number of memory ports can be increased.

As mentioned, we are currently using only one of the two FPGAs on the Gladiator board. Using both FPGAs is a promising option which would accelerate the rendering up to 2 times.

Furthermore, additional merging of algorithm functions would improve the performance. It would also increase data locality by means of a reduced memory transfer requirement. This would decrease the number of stall cycles and the memory bus traffic significantly.

Finally, a new version of the FTV algorithm is being developed by Eindhoven University. A speed-up by a factor of 2 can be expected by mapping this algorithm on our platform.

Resulting Schedules for the Warping Stage

A

In this chapter, we demonstrate some examples from schedules generated by the HiveCC compiler during a sched run. All the figures given indicates the cycle-accurate schedules for the inner loop of the warping function of the algorithm. By examining these schedules, we want to describe the affect of a number of optimizations on the performance of the warping stage. The vertical direction corresponds to the cycle number in the schedule, and the horizontal direction depicts the operations executed in corresponding issue slots. Note that we have 7 issue slot in total, the first two are scalar and the rest are vector issue slots. Furthermore, in all schedules, the starting and ending cycle count of the loop is given, and the latency-critical vamem load/store operations are marked with red circles.

Figure A.1 shows the schedule for our vectorized baseline with some optimizations (OP_vec_lod operation, data mapping) already applied. Please note that this schedule corresponds to only the warping of left depth map. The schedule for the warping of right depth map is the same. The schedule is very scarce due to the dependencies of the other computations on the vamem load/stores. The total schedule length is $2 \times 127 = 254$ cycles for the sum of the left and right warping. Since this belongs to processing a single vector (16 elements), this loop body takes $254/16 = 15,875$ cycles/pixel.

Figure A.2 depicts the schedule after applying loop merging on the previous schedule. In other words, this is the schedule corresponding to merging the left and right warping stage. The total schedule length is 150 cycles, which is equivalent to $150/16 = 9,375$ cycles/pixel.

By unrolling the loop (merged loops), we obtained the schedule shown in Figure A.3. Since an unrolling by a factor 2 is applied, this schedule corresponds to a loop body in which 2 vectors (32 elements) are processed. An increased parallelism capability enabled by loop unrolling, yielded a total schedule length of 218 cycles, which is equivalent to $218/32 = 6,8125$ cycles/pixel.

Finally, as mentioned in Section 5.2.1.5, the loop retiming technique is applied on the unrolled loop. The resulting schedule is very compact. The vamem load/store operations are now overlapped with the rest of the computations. The total schedule length of the inner-most loop is 142 cycles, corresponding to $142/32 = 4,4375$ cycles/pixel. As explained before, this is very close to the theoretical bound that can be achieved with the current memory configuration.

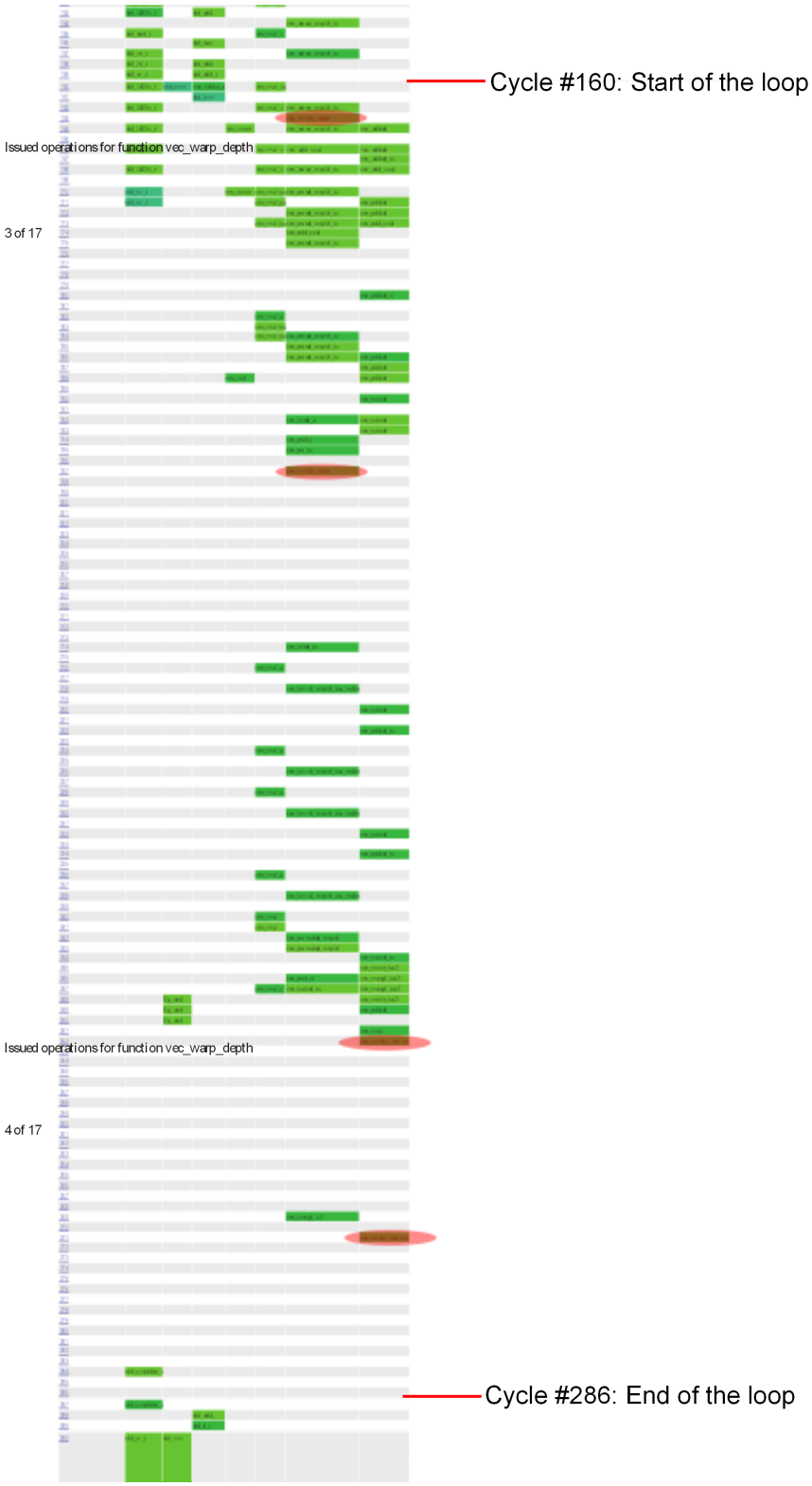


Figure A.1: The schedule for our vectorized baseline

Figure A.2: The schedule after applying the loop merging technique

Figure A.3: The schedule after applying the loop unrolling technique

Figure A.4: The schedule after applying the loop retiming technique

Bibliography

[1] D. Aliprandi and E. Piccinelli, *Real-time free viewpoint television for embedded systems*, Picture Coding Symposium (PCS), 2010, dec. 2010, pp. 346 –349.

[2] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the April 18-20, 1967, spring joint computer conference (New York, NY, USA), AFIPS '67 (Spring), ACM, 1967, pp. 483–485.

[3] Kubilay Atasu, Günhan Dündar, and Can Özturan, *An integer linear programming approach for identifying instruction-set extensions*, Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (New York, NY, USA), CODES+ISSS '05, ACM, 2005, pp. 172–177.

[4] Melvin C. August, Gerald M. Brost, Christopher C. Hsiung, and Alan J. Schiffleger, *Cray x-mp: The birth of a supercomputer.*, IEEE Computer.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp, *Compiler transformations for high-performance computing*, ACM Comput. Surv. **26** (1994), 345–420.

[6] Hendrik Boer, *Interactive free viewpoint 3d tv rendering platform*, Master's thesis, Eindhoven University of Technology, November 2010.

[7] E. Bondarev, R. Miquel, M. Imbert, S. Zinger, and P.H.N. de With, *On the technology roadmap of free-viewpoint 3dtv receivers*, Consumer Electronics (ICCE), 2011 IEEE International Conference on, jan. 2011, pp. 687 –688.

[8] David R. Butenhof, *Programming with posix threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] Silicon Hive B.V., *Silicon hive software development kit (sdk).*

[10] Consumer Electronics Association (CEA), *Cea 861- d : A dtv profile for uncompressed high speed digital interfaces*, July 2006.

[11] Hoseok Chang and Wonyong Sung, *Efficient vectorization of simd programs with non-aligned and irregular data access hardware*, Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems (New York, NY, USA), CASES '08, ACM, 2008, pp. 167–176.

[12] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy, *Dynamic scratch-pad memory management for irregular array access patterns*, Proceedings of the conference on Design, automation and test in Europe: Proceedings (3001 Leuven, Belgium, Belgium), DATE '06, European Design and Automation Association, 2006, pp. 931–936.

[13] Hsin-Jung Chen, Feng-Hsiang Lo, Fu-Chiang Jan, and Sheng-Dong Wu, *Real-time multi-view rendering architecture for autostereoscopic displays*, Circuits and Systems

(ISCAS), Proceedings of 2010 IEEE International Symposium on, 30 2010-june 2 2010, pp. 1165 –1168.

[14] Wan-Yu Chen, Yu-Lin Chang, Hsu-Kuang Chiu, Shao-Yi Chien, and Liang-Gee Chen, *Real-time depth image based rendering hardware accelerator for advanced three dimensional television system*, Multimedia and Expo, 2006 IEEE International Conference on, july 2006, pp. 2069 –2072.

[15] Andrea G. M. Cilio and Henk Corporaal, *Floating point to fixed point conversion of c code*, Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99 (London, UK), Springer-Verlag, 1999, pp. 229–243.

[16] Luat Do, Svitlana Zinger, and Peter H. N. de With, *Quality improving techniques for free-viewpoint dibr*, IST / SPIE Electronic Imaging, 2010, p. 10 pages.

[17] Luat Do, Svitlana Zinger, Yannick Morvan, and Peter H. N. de With, *Quality improving techniques in dibr for free-viewpoint video*, 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video, 2009.

[18] Chunyang Gou, Georgi Kuzmanov, and Georgi N. Gaydadjiev, *Matched sams scheme: Supporting multiple stride unaligned vector accesses with multiple memory modules*, Tech. Rep.

[19] Silicon Hive, *The silicon hive company webpage*, Online, 2011, Availabe at: http://www.siliconhive.com.

[20] The iGLANCE Project, *The iglance project webpage*, Online, 2011, Availabe at: http://iglance.org.

[21] Mahmut Kandemir, N. Vijaykrishnan, and Mary Jane Irwin, *Compiler optimizations for low power systems*, pp. 191–210, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[22] K. Keutzer, S. Malik, and A.R. Newton, *From asic to asip: the next design discontinuity*, Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on, 2002, pp. 84 – 90.

[23] Leonard McMillan, Jr., and Reader Stephen Pizer, *An image-based approach to three-dimensional computer graphics*, Tech. report, 1997.

[24] Yuji Mori, Norishige Fukushima, Tomohiro Yendo, Toshiaki Fujii, and Masayuki Tanimoto, *View generation with 3d warping using depth information for ftv*, Image Commun. **24** (2009), 65–72.

[25] Dorit Naishlos, Marina Biberstein, Shay Ben-David, and Ayal Zaks, *Vectorizing for a simdd dsp architecture*, Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (New York, NY, USA), CASES '03, ACM, 2003, pp. 2–11.

[26] M.M. Oliveira, *Relief texture mapping*, Ph.D. thesis, University of North Carolina, 2000.

[27] Learning Opengl, Silicon Graphics, and The Silicon Graphics Logo, *Opengl programming guide (addison-wesley publishing company) second edition opengl programming guide (addison-wesley publishing company) the official guide to.*

[28] Nelson Luiz Passos and Edwin Hsing-Mean Sha, *Full parallelism in uniform nested loops using multi-dimensional retiming*, Proceedings of the 1994 International Conference on Parallel Processing - Volume 02 (Washington, DC, USA), ICPP '94, IEEE Computer Society, 1994, pp. 130–133.

[29] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, and Werner Stuetzle, *View-based rendering: Visualizing real objects from scanned range and color data*, In Eurographics Rendering Workshop, 1997, pp. 23–34.

[30] Jaewook Shin, *Introducing control flow into vectorized code*, Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (Washington, DC, USA), PACT '07, IEEE Computer Society, 2007, pp. 280–291.

[31] Hock Soon Tan, Jiazhi Xia, Ying He, and YQ Guan, *A system for capturing, rendering and multiplexing images on multi-view autostereoscopic display*, Cyberworlds, International Conference on **0** (2010), 325–330.

[32] Pei-Kuei Tsung, Ping-Chih Lin, Kuan-Yu Chen, Tzu-Der Chuang, Hsin-Jung Yang, Shao-Yi Chien, Li-Fu Ding, Wei-Yin Chen, Chih-Chi Cheng, Tung-Chien Chen, and Liang-Gee Chen, *A 216fps 4096×2160p 3dtv set-top box soc for free-viewpoint 3dtv applications*, Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International, feb. 2011, pp. 124 –126.

[33] Fan Zhang and Yuli Xu, *Image quality evaluation based on human visual perception*, Proceedings of the 21st annual international conference on Chinese Control and Decision Conference (Piscataway, NJ, USA), CCDC'09, IEEE Press, 2009, pp. 1542–1545.

[34] Svitlana Zinger, Luat Do, Daniel Ruijters, and Peter H. N. de With, *Iglance: interactive free viewpoint for 3d tv*, 3D Stereo Media conference, 2009, p. 4 pages.

[35] Svitlana Zinger, Daniel Ruijters, and Peter H.N. de With, *iglance project : free-viewpoint 3d video*, international conference in Central Europe on computer graphics, visualization and computer vision (2009), 35 –38.

[36] C. Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski, *High-quality video view interpolation using a layered representation*, ACM Trans. Graph. **23** (2004), 600–608.