Hardware Acceleration of the Trace Transform for Vision Applications

Suhaib A. Fahmy

This thesis is submitted for the degree of Doctor of Philosophy of the University of London and for the Diploma of Imperial College

Department of Electrical and Electronic Engineering

Imperial College London

University of London

December 2007

Abstract

Computer Vision is a rapidly developing field in which machines process visual data to extract meaningful information. Digitised images in their pixels and bits serve no purpose of their own. It is only by interpreting the data, and extracting higher level information that a scene can be understood. The algorithms that enable this process are often complex, and data-intensive, limiting the processing rate when implemented in software. Hardware-accelerated implementations provide a significant performance boost that can enable real-time processing.

The Trace Transform is a newly proposed algorithm that has been proven effective in image categorisation and recognition tasks. It is flexibly defined allowing the mathematical details to be tailored to the target application. However, it is highly computationally intensive, which limits its applications. Modern heterogeneous FPGAs provide an ideal platform for accelerating the Trace transform for real-time performance, while also allowing an element of flexibility, which highly suits the generality of the Trace transform. This thesis details the implementation of an extensible Trace transform architecture for

vision applications, before extending this architecture to a full flexible platform suited to the exploration of Trace transform applications. As part of the work presented, a general set of architectures for large-windowed median and weighted median filters are presented as required for a number of Trace transform implementations. Finally an acceleration of Pseudo 2-Dimensional Hidden Markov Model decoding, usable in a person detection system, is presented. Such a system can be used to extract frames of interest from a video sequence, to be subsequently processed by the Trace transform.

All these architectures emphasise the need for considered, platform-driven design in achieving maximum performance through hardware acceleration.

Acknowledgements

I would like to thank my supervisor, Professor Peter Cheung for his guidance and support throughout this PhD. He has often gone out of his way to offer support outside of his academic duties, and for this I am grateful. I am also indebted to Dr. Christos Bouganis for his time, advice and mentoring through the last two years. Professor Wayne Luk has provided significant contributions and feedback for my publications and thesis. I would also like to thank Pete Sedcole for being there when I needed to bounce ideas off someone, and providing advice during the writing of this thesis. I received much appreciated counsel from many sources during the write-up as well as the work itself. Thanks to Alastair Smith, Ben Cope, Altaf Gaffar, Sherif Yusuf, Jon Clarke, Kieron Turkington, George Constantinides, Nick Campregher and others I may have failed to mention by name.

Numerous fellow PhD graduates have been there for me throughout with words of encouragement and support when writing was tough and when the viva was looming. I give a heartfelt thanks to Kashif Javaid, Ismail Jalisi, Khurom Kiyani and Mahbub Gani. And those who are yet to reach the final

destination, Kareem Osman and Aswad Manzoor, I wish them luck. Few days will be remembered as fondly as my days at Imperial.

I cannot fail to thank my beloved wife for her patience and support over three and a half years of dealing with PhD Syndrome. My three wonderful children, Hafsah, Talhah and Yahya, have interspersed this PhD with moments of sheer joy and happiness. I apologise for the times I couldn't be there, even if, perhaps, I was in body.

I also owe a debt of gratitude to my parents for their support and prayers throughout my studies and before. And my dad for understanding the pressures of writing-up having done it himself 28 years ago. I also thank my in-laws for their support throughout.

Yet above and beyond all, and without Whom there would be no-one to thank in the first place, I praise God, the Almighty and thank Him for surrounding me with such great people and providing me with opportunities, endless as they are. There are many, the world over who will never taste these opportunities and who will pass away having never felt fulfilment. May we always be grateful for what we have, and show compassion for those less fortunate than ourselves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Humans interact daily with their surroundings, through perception via our senses and physical interaction with our limbs. When a sensing faculty is disabled, or its ability diminished, one's perception of his surroundings decreases, and so he acts with more uncertainty. As humans, we are able to process the boundless information that we receive, especially visually, to build some understanding of the world around us. Images in their pixels and bits serve no specific purpose. It is only through extracting higher-level information that some understanding of the scene in question can be gained.

Recently, much effort has been invested in giving machines the ability to interpret visual data. Computer vision is a fast-moving field with many exciting developments. A typical computer vision processing flow is shown in Figure 1.1. Image data is first processed in order to extract features that can be used to represent the image; perhaps edges, corners or other features. These can then be processed to detect the presence of an object. A detected

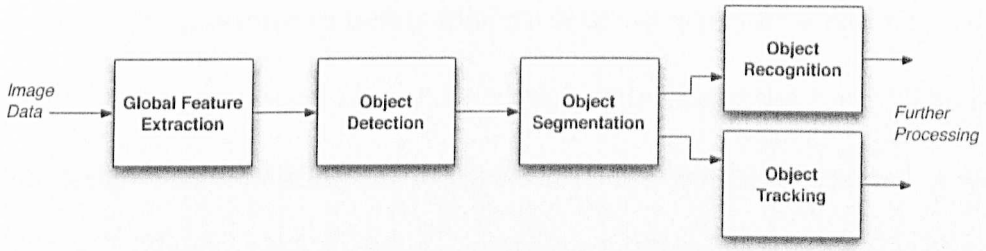


Figure 1.1: A typical computer vision processing flow.

object can be segmented; effectively "cutting it out" of the scene. This object might then be processed for authentication purposes or for temporal tracking or other higher-order tasks.

Computer vision applications are often characterised by the large amounts of data and processing needed to implement them. The most effective algorithms have often also proven to be some of the most complex. Such algorithms often fail to achieve real-time performance in software and so must be accelerated somehow in order for them to be of significant use. Designing efficient hardware implementations can often propel these systems to the realms of real-time performance. Hardware design is a complex process, and historically, it has been out of the reach of most. This has, however, begun to change recently.

Field Programmable Gate Arrays (FPGAs) are an emerging force in the hardware arena, offering some of the power of custom-designed hardware for a fraction of the effort and cost. Their relative ease of design, low starting cost, rapid time to market and re-configurability make them both an excellent prototyping platform and an ideal alternative to Application Specific Integrated Circuits (ASICs) for medium volume applications. The most significant benefits in any hardware implementation are gained when hardware is designed

with the target platform in mind. Modern heterogeneous FPGAs offer a wide array of resource types which can be exploited in numerous ways. The skill of the designer in exploiting the available resources to their potential is what separates a mediocre system from an efficient, significantly accelerated architecture.

Such a platform provides all the ingredients needed for computer vision research and implementation. FPGAs afford the designer the opportunity to research different algorithms, to incrementally improve implementations and to test applications in the field without huge start-up costs and the associated risks.

The Trace transform is a recently introduced algorithm that has been shown to perform well in a variety of image recognition and categorisation tasks. It maps a standard image to an alternative domain, and while defining the spatial mapping, is general in terms of the mathematical aspect. This allows the transform the flexibility to adapt to different applications and for the mathematical components to be selected with respect to their performance for a specific task. The Trace transform is, however, computationally intensive, and acceleration would enable real-time performance that is as yet unachieved in software.

In a vision flow, the Trace transform can be used in numerous ways. It is possible to use the Trace transform to extract global features from an image. These features can be used to characterise certain aspects of the image. An example is the car park usage classification system discussed in Section 3.5.3.

The Trace transform has thus far been primarily used for object recognition or authentication, which would typically follow an object segmentation step. The Trace Transform has been applied to image database search [KP01] and face authentication [SPKK05], covered in more detail in Sections 3.5.1 and 3.5.4, respectively.

One of the Trace transform's strengths is its flexibility in terms of the computational mapping. This flexibility is essential in optimally applying the transform to a desired application. Hence, this thesis details an architecture that maintains flexibility and scalability. One of the oft-used mathematical functions within these arithmetic blocks is the median and weighted median. Developing an efficient architecture to implement this enables the use of some of the more complex arithmetic mappings.

Pseudo 2-dimensional HMM decoding is a method that is useful in detection and recognition tasks. The work in this thesis is related to an application used for person tracking [BR03]. Similar systems have been used for face detection [Nef99]. The pseudo 2-dimensional HMM can be used as a stage prior to the Trace transform, that extracts an object of interest for processing by the transform.

This thesis will investigate the use of FPGAs in computer vision, with the primary focus being the hardware acceleration of the novel Trace transform. All aspects of the implementations will be discussed from design methods, through architectural considerations, down to the implementation results. Beside real-time acceleration, the architecture will be extended to allow for a

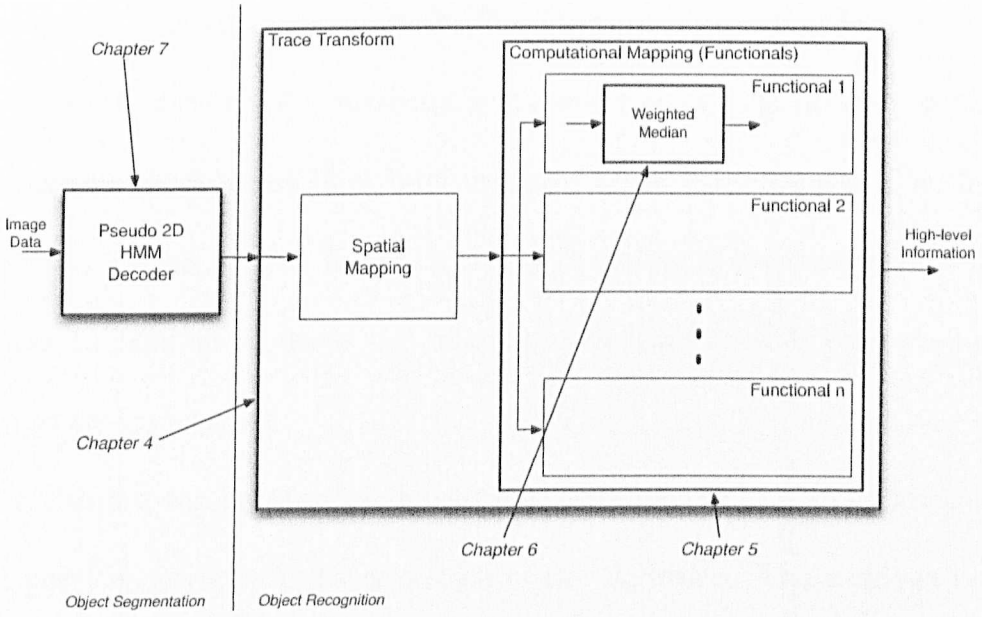


Figure 1.2: A vision flow incorporating the work in this thesis.

fully flexible set of mathematical units, as well as a framework for creating such units and reconfiguring them at runtime. All the architectures presented emphasise the need for considered design and the exploitation of heterogeneous resources to achieve optimum performance.

An overview of a vision flow incorporating the work in this thesis is shown in Figure 1.2.

1.1 Thesis Outline

The remainder of this thesis is composed of seven further chapters as follows:

Chapter 2 covers general background information on Field-Programmable Gate Arrays, the architectures, design processes and performance metrics used to measure implementations. A general overview of the computer vision domain is given with a summary of FPGA implementations of various algorithms.

Chapter 3 is an introduction to the Trace transform and its precursor, the Radon transform. The mathematical foundations of the Radon transform are introduced, followed by a brief overview of its applications. The Trace transform is then defined before looking at a variety of applications and discussing its different modes of use. Finally, some analysis of its computational complexity is presented.

In Chapter 4, a hardware architecture is developed for the Trace transform that provides for real-time acceleration of the algorithm. The architecture is detailed showing how significant acceleration is achieved through the exploitation of algorithmic parallelism. The architecture presented is extensible, and can be used to build up a full recognition system using the Trace transform.

In Chapter 5, a framework for developing flexible, re-programmable functionals is presented. The framework interfaces with the architecture developed in Chapter 4, and while adding significant flexibility, has no adverse impact on performance. As a reference, three flexible functional blocks are implemented, each with the capability to compute a number of different functionals from an existent implementation. The framework facilitates the exploration of Trace transform functionals for a given application.

In Chapter 6, a highly efficient hardware implementation of large-windowed median and weighted median filters is developed. This implementation assists in implementing some of the more complex Trace transform functionals in a real-time system. Numerous design variations including fixed vs. sliding windowed, fixed vs. variable median index and standard vs. weighted median

are investigated in terms of their area usage.

In Chapter 7, a hardware architecture for Pseudo 2-dimensional Hidden Markov Model (HMM) decoding, as used in a person detection system, is presented. Such a system can serve as an initial step in a full person recognition system, extracting frames of interest from a video stream which can then be processed using the Trace transform. The HMM decoding is accelerated by analysing the state transitions and optimising the hardware accordingly, while also exploiting algorithmic parallelism. The architecture is generalisable to any pseudo-2D HMM.

Finally, in Chapter 8, the work is summarised, along with the conclusions reached from these implementations. Finally, some suggestions for future work are given.

1.2 Contributions

The main contributions of this thesis are as follows:

- A thorough computational analysis of the Trace transform, including a look at areas of algorithmic parallelism that can be exploited for hardware acceleration. (Section 3.6).

- The first hardware implementation of an extensible Trace transform architecture, which achieves real-time processing speeds, while remaining fully flexible in terms of the number of functionals implemented. A novel approach to parallelising rotations through the concatenation of

orthogonal base rotations quadruples performance while designing the architecture to process a stream of image data removes the need for any internal buffering. (Chapter 4).

- A framework for developing flexible functionals for use in the Trace transform architecture mentioned above, including three examples. Embedded memories on the FPGA are used for function evaluation, and multiple datapaths are selectable via a configuration register, thus allowing a single functional block to compute a range of different functional equations. This framework serves as an ideal platform for further investigation of the Trace transform itself for a variety of applications. (Chapter 5).

- A highly flexible set of architectures for implementing one-dimensional large-windowed median and weighted median filters for image processing. A rank of cumulative histogram bins is addressed in parallel, keeping a fully updated cumulative histogram with every sample that enters the system. The architecture is unique in its ability to process windows of arbitrary size, and without an area increase. The architecture implements both standard and weighted median calculation for fixed and variable windows. (Chapter 6).

- A real-time acceleration of Pseudo 2-dimensional Hidden Markov Model decoding. By considering the state transition trellis for a pseudo 2-dimensional HMM, significant simplifications can be made to the decoding stage. This simplifies the otherwise complex Viterbi calculation used

in the system. The pseudo 2D HMM has been shown to be applicable to image segmentation as part of an object tracking system. (Chapter 7).

1.3 Publications

Parts of the work detailed in this thesis have also been separately published in the following publications:

- "Hardware Acceleration of Hidden Markov Model Decoding for Person Detection" S.A. Fahmy, P.Y.K. Cheung, W. Luk. Proceedings of Design, Automation and Test in Europe (DATE), 7-11 March 2005, Munich, Germany. Volume 3, Pages 8-13. [FCL05a]

- "Novel FPGA-Based Implementation of Median and Weighted Median Filters for Image Processing" S.A. Fahmy, P.Y.K. Cheung, W. Luk. Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 24-28 August 2005, Tampere, Finland. Pages 142-147. [FCL05b]

- [FBCL06]: "Efficient Realtime Implementation of the Trace Transform" S.A. Fahmy, C.-S. Bouganis, P.Y.K. Cheung, W. Luk. Proceedings of International Conference on Field Programmable Logic and Applications (FPL), August 2006, Madrid, Spain. [FBCL06]

- "Real-Time Hardware Acceleration of the Trace Transform" S.A. Fahmy, C.-S. Bouganis, P.Y.K. Cheung, W. Luk. Journal of Real-Time Image

Processing: Special Issue on Field-Programmable Technology, Springer, December 2007. [FBCL07]

- "From Algorithms to Architecture" S.A. Fahmy, C.-S. Bouganis, P.Y.K. Cheung. Chapter 11 of A. Bharath and M. Petrou (editors) Reverse Engineering the Human Vision System, Artech Publishers, to appear in 2008. [BM08]

Chapter 2

Background

2.1 Introduction

Field Programmable Gate Arrays (FPGAs) are a relatively recent development when considered against the backdrop of decades of development in the field of digital electronics. The simple digital circuits of yesteryear were often constructed using off-the shelf logic devices that could manage only a very primitive, single logic operation each. A board with hundreds of these small chips might implement an archaic system of minimal complexity. Only the most well-funded corporations could afford to design their own devices from the ground up, using what were the cutting edge Computer-Aided Design (CAD) tools of their time, and the newly emerging integrated circuit (IC) technologies. For the hobbyist or small company or research effort, these "advanced" technologies were out of reach.

The emergence of ICs, driven initially by some high-profile aerospace projects

saw the start of a rapidly developing landscape of custom designed integrated circuits, offering high performance and compactness. The growth has been exponential, and today, some Application Specific Integrated Circuits (ASICs) contain hundreds of millions of transistors on-die.

In this chapter, background information on FPGAs will be provided, with some insight into the design process and various design decisions that must be made.

A review of some hardware implementations of computer vision applications will also be presented. Computer vision systems are computationally very complex, and hardware implementations are often required for real-time performance. Background related to the Trace transform will be presented in the next chapter.

2.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) were invented in 1984 by Ross Freeman [Xil04], one of the founders of Xilinx Inc. In 1985, Xilinx released their first FPGA, the XC2064 which contained 64 logic blocks and 1000 gates [Xil04]. Today, FPGAs have become a viable platform for implementing some of the most complex digital designs with sizes the equivalent of tens of millions of gates. Whereas FPGAs were seen as a platform for implementing glue-logic in the early days, they now find use in a wide range of applications, and often feature in end-market products.

When choosing an implementation platform for a given application or product, the typical choice one would make is between custom ASIC, some form of Application-Specific Standard Platform (ASSP), such as Digital Signal Processors (DSPs) or Graphics Processing Units (GPUs) and standard General Purpose Processors (GPPs). GPPs are simpler versions of what one might find in his PC; simple microprocessors that can compute a wide variety of different functions by breaking them up into standard instructions that are manageable by the on-die resources. Typically, an application would be programmed in a familiar high-level language and a compiler would then translate this into native "machine-code". ASSPs are processors suited to a specific implementation domain; typically the on-die pipelines and processing units are tailored to the needs of specific tasks required by the target domain. DSPs (e.g. Texas Instruments TMS320 series) are the most prevalent and typically allow for real-time implementation of complex signal processing applications. GPUs (e.g. ATI Radeon series) are highly optimised for graphics tasks, and indeed the transistor count of some GPUs exceeds that of many top-end GPPs. Network processors are another form of ASSP that has become widely used in the field of networking. ASSPs are also typically programmed using high-level languages, sometimes with extensions specific to the application domain.

There is little doubt that for the highest possible performance for a given, fixed application, that custom ASICs are the platform of choice. A custom ASIC is designed from the ground up to implement the specified application. A hardware architecture tailored to the system at hand is developed and every-

thing is tweaked to the parameters of the application. The significant speedup results from two things:

Firstly, within most complex algorithms, there is some complex arithmetic computation. When such computations are run on general purpose processors, the steps must be broken down into small instructions that can be executed by the pipeline present on the target device. Standard programming languages make this easier by allowing a coder to specify complex instructions, while the compiler breaks these up into "machine code". Typically, a general purpose processor will have a core processing unit that is capable of computing a fixed variation of simple operations at very high speed. While the number of computations per second may be very high, the number of complete operations per second is significantly reduced. This is especially pronounced when the operations being performed are far from those intended by the design of the pipeline. This is what has pushed the development of ASSPs.

When designing a hardware system, the designer is freed from these constraints, since the computational unit can be tailored to specified requirements, and the designer is free to implement any number of different units. By implementing a core that is custom-designed to implement specific operations, the core can run slower (in terms of clock speed) than a general purpose processor but still have significantly greater processing throughput[1].

Secondly, an algorithm can be accelerated in hardware through parallel

[1]Throughput refers to the real overall processing speed of a system, and is typically measured in full data units completed per second. For image and video processing, the typical measure would be the number of frames processed per second.

computation. Complex algorithms typically iterate over various values of parameters in order to compute one complete result. As an example, a 2-dimensional image filtering algorithm would require iterations along all pixels in the x and y axes. Often there is somewhere in the algorithm where such a set of iterations occurs. In a hardware system, the designer is free to implement as many computational blocks as the resources will allow, and this means that multiple serial iterations can in fact be processed simultaneously. The only limitation is that the results for each iteration of a variable must be independent of each other: the calculation of results for one iteration should not depend on the results of another. When this is the case, the designer can design the system to exploit this algorithmic parallelism in order to afford significant speedup.

As a result of this custom tailored design, the resultant hardware is optimised for speed, area and power, and significantly outperforms the equivalent software system running on a GPP or ASSP. Of course, there are other factors to be considered which reduce this advantage somewhat. Firstly, the design and verification of a custom ASIC is a complex, time-consuming process. As such, the non-recurring engineering (NRE) costs are significant. Couple this with the initial costs of production, which can exceed $1 million in the case of some of the newer manufacturing technologies, and the long time-to-market and it is clear that economies of scale play a big part in the viability argument. Furthermore, one may wish to consider the increasing rate at which new standards are being introduced. An ASIC implementation is fixed, and thus any

envisaged changes in the design would either have to be incorporated into the initial architecture, or else, a new chip would have to be produced, with the associated costs.

FPGAs can be seen as a half-way house between custom ASICs and ASSPs. They offer the significant performance advantage of a custom-designed architecture, with much lower NRE and implementation costs than ASICs; they are available as off-the-shelf products. FPGAs are reconfigurable, which means that an implemented design can be changed or replaced, even after system integration. There is no need for low-level verification of a design, since FPGAs are designed to meet specified constraints for on-chip logic and I/O. Of course, there are still other factors to consider in deciding whether to use FPGAs for a specific application. Firstly, an FPGA implementation is still a hardware design, and hence the required knowledge is arguably more specialised than that needed for software implementations. Furthermore, the cost of FPGAs is higher than ASICs for very large quantities. One of the common examples where an ASIC implementation is more enticing is that of codec chips, such as an MPEG-4 decoder. Given a fixed design that will be used in hundreds of millions of devices, across an array of applications, the cost of an ASIC implementation becomes lower than using FPGAs. FPGAs also typically consume more power than equivalent designs in ASIC, so are generally not favoured in mobile applications.

As technology has improved, however, many of the disadvantages of using FPGAs have been tackled. While much benefit has been gained from general

Platform	Advantages	Disadvantages
GPP	Very simple to program, widely available, can be distributed as standard software.	Poor performance, very limited architecture.
ASSP	Simple to program, tailored to application domain, with excellent performance for the specified applications.	Available for specific applications only, performance is not as fast as custom ASIC for a non-standard application.
FPGA	High performance, significant flexibility, design effort less than ASIC.	Design more complex than software, high power consumption.
Custom ASIC	Best performance available, completely tailored to application.	Very high costs, complex design and verification, fixed design cannot be changed.

Table 2.1: Advantages and disadvantages of various target platforms.

advancements in fabrication processes, a number of functional improvements have seen the FPGA become a more viable target platform for a plethora of application domains. The latest generations of FPGAs have significantly reduced power requirements, are able to run at higher speeds, support a wider array of I/O standards, and offer significantly more complex on-die units. Couple this with the rising costs of implementing cutting-edge ASICs and the rapid development and deployment of new industry standards, and it is clear that the advantages held by FPGAs in terms of rapid design time, and deployment, with reduced time-to-market make them an attractive alternative to ASICs. Some of the trade-offs in selecting a target platform are summarised in Table 2.1.

2.2.1 Logic and Routing

The principle behind an FPGA is an architecture that can implement any arbitrarily defined function. The FPGA is a pre-fabricated circuit, needing nothing more than a simple configuration to be applied in order for it to function. For the flexibility to implement any function, two requirements must be met. Firstly, there must be simple, flexible circuit elements that can implement arbitrary logic functions; this is often termed the logic fabric. Secondly, there must be some method of connecting such units up in an arbitrary fashion, the routing fabric. These elements are the core of FPGA architectures.

The logic fabric is typically composed of circuit elements built around small Look-Up Tables (LUTs). These can perform any given logic function with a single output. A 4-input LUT can implement any logic function of four variables with a single output. The output of this LUT is typically connected to an optional flip-flop allowing for synchronous circuits. The routing fabric connects multiple logic elements together through switch-boxes. These are separately programmable to connect arbitrary logic elements together. This flexibility is the key to an FPGA's reprogrammability, as well as its ability to implement any design. Both the contents of the LUTs and the routing switches are reconfigurable.

Of course, this level of flexibility comes as a cost. Whereas in an ASIC design, fixed wires route between circuit elements, in an FPGA, signals must travel through the routing fabric including switches and drive wires which may

be longer than are absolutely necessary. This all introduces delay, creating a performance gap between an FPGA and ASIC implementation of the same design. One must also consider the fact that a design is targeted to an FPGA with a specified number of logic-elements: the sizes step-up in stages, and so it is likely that some part of the FPGA's logic fabric will be unused in a design. Enabling the huge number of possible connections in an FPGA architecture also means that the routing is very abundant, which means that a significant portion of chip area is consumed by non-computational elements.

The earlier FPGA architectures, such as the Xilinx 4000 series [Xil99a] and the Altera FLEX 8000 [Alt01b] consisted solely of the logic elements and routing fabric as described above, as well as I/O blocks for off-chip communication. These architectures developed rapidly, with the addition of carry-chain logic to the the basic elements and other tweaks to the logic fabric to allow for more efficient implementation of common design components. The delay-cost of routing has driven significant changes in the routing fabric too, with more considered (and complex) routing arrangement including hierarchical routing and multiple wire lengths. The most recent FPGA devices have departed from 4-input LUTs to 6-input LUTs (in the case of the Virtex 5 [Xil07b]) and adaptive LUTs and adders (in the case of the Stratix III [Alt07]).

2.2.2 Reconfigurability

FPGAs are SRAM-based devices. The logic and routing are configured at runtime, and upon power-off, this configuration is lost. This is why a produc-

tion FPGA will typically have a Programmable ROM (PROM) that stores the configuration data sited beside it, so that it can configure when it powers-up. This also means that the system configuration can be changed at any time, and the circuit can be modified. Typically, this might be used to try different variations of a circuit or fix problems that arise in simulation. However, another possibility is what has been termed dynamic reconfiguration. This is where the FPGA configuration is modified while it is running. Partial reconfiguration is where only a part of the circuit is modified at runtime, perhaps to implement an alternative block within the same architecture. A thorough explanation is given in [Sed06].

Unfortunately, the design flow for a reconfigurable architecture is significantly more complex than that for a static architecture. At present, the designer needs to work at a relatively low level, managing the placement of reconfigurable blocks, in order to allow for partial reconfiguration. This situation is changing but as yet, this field remains the preserve of academic research.

2.2.3 Embedded Memories

A significant development to FPGA architectures was the addition of other types of resources such as embedded memories, as in the case of the Xilinx Virtex [Xil99c] and Altera FLEX 10K [Alt01a]. If one considers the logic fabric mentioned above, it is clear that implementing memories of any reasonable size would be highly inefficient in LUTs. Memories are useful in many applications, as buffers, FIFOs and for temporary storage. Small embedded memories negate

the need for routing between LUTs, thus increasing the speed, and reducing the area and power consumption of memory accesses. This also saves on the need to use off-chip RAM, and thus the costs associated with I/O. The Xilinx Virtex brought between 8 and 32 4Kbit RAM blocks per device (depending on device size). Later, the Xilinx Virtex II [Xil99b] saw these enlarged to 18Kbits each as well as increasing in number. The Virtex 4 [Xil07a] maintains the same Block RAM configuration, while the Virtex 5 [Xil07b] increases the capacity to 36kbits each. RAMs can also be combined without the use of extra logic in the Virtex 4 and Virtex 5. Altera, on the other hand, has developed a hierarchical memory architecture for its Stratix [Alt05] series, with three different sizes of RAMs on chip. This suits applications where a few large buffers might be needed, as well as small local coefficient memories.

Embedded memories have significant advantages. A 4-input LUT can only implement a 16x1bit ROM. Thus for any reasonable sized memory, a significant number of Slices and consequently routing resources would be required. As a comparison, a 16k×1bit ROM, implemented in logic on a Xilinx Virtex II would use 559 Slices and could only be clocked at 2/3 the speed of the equivalent implemented in a Block RAM [Smi07]. The Block RAMs on the Xilinx Virtex II can be implemented as single- or dual-port ROMs or RAMs, synchronous or asynchronous FIFOs and also data width converters [Xil99b].

2.2.4 Embedded Multipliers and DSP Blocks

The Xilinx Virtex II brought with it embedded multipliers. FPGAs had found a significant following in the DSP community and multipliers are a feature of many DSP designs. Since implementing multipliers using LUTs carries with it all the inefficiency of the routing between LUTs, hard-wired multipliers can free up significant resources for other tasks. The Altera Stratix pushed this idea further by implementing a DSP block, another name for a multiply-accumulator (MAC). The latest Xilinx Virtex 4 and 5 [Xil07a, Xil07b] have followed suit.

Multipliers find their most obvious use in digital filters. In the Virtex II, the architecture was designed with the Block RAMs beside the multipliers so as to minimise routing delays between the coefficients, typically stored in memories, and the multipliers. Aside from traditional uses, multipliers have also been used as barrel-shifters [Gig04], for implementing floating-point units and even to replace Block RAMs [MCC07]. In fact, multiplications surface in a significant number of image processing applications from colour-space conversion and image rotation, to filtering and image transforms (see [Rus02] for an overview of algorithms). The saving in using embedded multipliers is both in terms of logic fabric and routing resources. An 18×18 bit multiplier implemented in logic on a Xilinx Virtex II would use 201 Slices and would only run at half the speed of the equivalent embedded multiplier [Smi07].

2.2.5 Other Resources

Modern FPGAs include a number of other resources which find uses in various applications. The Xilinx Virtex II Pro introduced embedded PowerPC microprocessors. These are offered on some parts of the new Virtex 4 and 5 families today. Having a processor on chip means that a whole software/hardware solution can be developed on a single chip. This is often termed System-on-a-Programmable-Chip (SoPC) development. The PowerPC can even been used to run an embedded version of Linux on the FPGA [Sai04]. It is also possible to allow the PowerPC to control runtime reconfiguration of the FPGA [BJRK+03]. In those FPGAs where no processor is available, vendors offer a soft processor that can be implemented using the logic fabric. Xilinx offers the MicroBlaze and PicoBlaze, while Altera offers the NIOS processor.

Delay-Locked Loops (DLLs), Phase-Locked Loops (PLLs) and Digital Clock Management (DCM) blocks allow fine control over clock signals, including the facilitation of multiple clock domains in a design. Advanced I/O standards are also supported on some devices, with gigabit transceivers and differential signalling built in to the fabric. I/O is one area where FPGAs shine. Since the I/Os are verified against multiple standards one need not concern themselves with this process, as would be the case with an ASIC design.

2.2.6 The FPGA Design Flow

The FPGA design flow is similar to that of standard ASIC hardware, but with some notable differences due to the different target platform. Here, the various steps are detailed along with a discussions of some of the design decisions that can be taken.

Design Entry

The first step in the flow is design entry. This is where the hardware architecture is specified. This can be done at a number of different levels of abstraction, and using various different tools and languages. One of the more archaic methods is Schematic Capture. Typically a computer-based design program, that contains visual representations of basic building blocks, is used. The designer places these on a canvas, defines the parameters of each of the blocks then connects them graphically, as required. The designer then defines the inputs to the system and its outputs, and the tools take care of the rest. However, with the increase in design complexity, this method of design entry has become more rarely used.

Another method is the use of Hardware Description Languages (HDLs). These are special languages used to describe hardware. Initially, they were used to describe a circuit structurally, in terms of its low-level components. These languages can also be used to describe hardware at Register Transfer Level (RTL), where the system is designed in terms of a set of registers and transfer functions describing the flow of data between them. Now, a more

significant portion of designs are described at a behavioural level. At this level of abstraction, the designer specifies what the circuit does, leaving the synthesis tools to determine how to implement this behaviour. As an example, the designer can simply specify $A+B=C$, and the compiler will determine the circuitry to do this. This allows the designer to focus less on the small details, and also means that the same code can be used to target multiple architectures using appropriate synthesis tools. The foremost HDLs used today are VHDL and Verilog.

Recently, significant effort has been spent in developing tools that allow the designer to work at higher levels of abstraction. The idea behind "High-Level Synthesis" is that the designer should be free to focus on the system-level design and ignore the details of the hardware implementation. Extensions to the C programming language, such as Handel-C [Cel], add language elements for hardware description yet allow the designer to use familiar C-based syntax. Xilinx's System Generator [Xil] is a tool that latches into Mathworks' Simulink software [Mat], used in DSP design. It allows the hardware designer to draw dataflow graphs to describe a DSP system, and the tools take care of the translation to hardware. High-level synthesis, however, is still relatively novel, and the performance of most tools cannot compete with the performance and compactness of systems designed at lower levels of abstraction. Vendor tools such as System Generator are significantly better than general High-level languages, since often, they contain pre-defined blocks that have already been optimised for hardware implementation. Such tools though, are

often restricted to a specific application domain; in the case of System Generator, DSP systems. True general purpose high-level synthesis is much more difficult to achieve, and hence cannot always be considered as the ideal design tool. With languages like Handel-C, the added language constructs still allow the designer to design at a level of abstraction similar to behavioural HDL descriptions, and when used in such a manner, the resultant implementations are significantly faster and more compact than when the language is used in a software-centric way. [CH02] and [TCW+05] both discuss the various design tools and descriptions available and the trade-offs associated with high-level synthesis. This, however, remains a fast-developing field and the situation continues to improve. A full scientific study of the trade-offs associated with different design description would be a welcome resource to assist designers in selecting the most suitable tools.

Notes on Handel-C

Since some of the work in this thesis was completed using Handel-C, it is worth noting some of the features of the language and design environment. Handel-C extends a subset of the C language to allow for customisable data widths and parallelism; two essential elements of hardware design. Timing is fixed at one clock cycle per C statement, allowing the designer fine control over circuit scheduling. The compiled circuit is a one-hot state machine that uses token-passing to move from one statement (or a block in the hardware) to the next.

The design environment for Handel-C is the Celoxica DK suite, which provides a code editor and simple simulator. The simulation tools can be awkward for hardware design since concurrency is difficult to track using a variable watch window. The real strength of using the Celoxica tools comes in the board APIs, that enable the use of development board resources with minimal effort, abstracting complex control and data signalling to simple C statements.

Handel-C code is compiled to an EDIF netlist or to VHDL code, that can then be further synthesised, placed and routed using the standard tools described below. Due to the hidden control circuitry, there is an area and speed overhead, but no concrete figures are available, since this depends entirely on the specific application. Massively parallel systems, such as the median calculation architecture presented in Chapter 6 suffer more than small regular circuits or those with complex control, but only a few parallel blocks.

Coding in Handel-C can be simpler than VHDl for complex designs. There are numerous constructs that assist in code reuse, and replication. Furthermore, since the clocking is inherent in the code, it can be tidier. Channels allow for timing-blind inter-block communication and synchronisation with ease. Figures 2.1 and 2.2 show a simple block of code in both VHDL and Handel-C (respectively), with the resultant circuit shown in Figure 2.3.

Functional Verification

In this step of the design flow, the designer confirms that the circuit, as described, implements the desired behaviour. A testbench is written, that wraps

```
data_proc: process(clk)
begin
    if rising_edge(clk) then
        a <= ain;
        b <= bin;
        x <= a + b;
    end if;
end process data_proc;
```

Figure 2.1: Example VHDL Code.

```
par {
    a = ain;
    b = bin;
    x = a + b;
}
```

Figure 2.2: Example Handel-C Code.

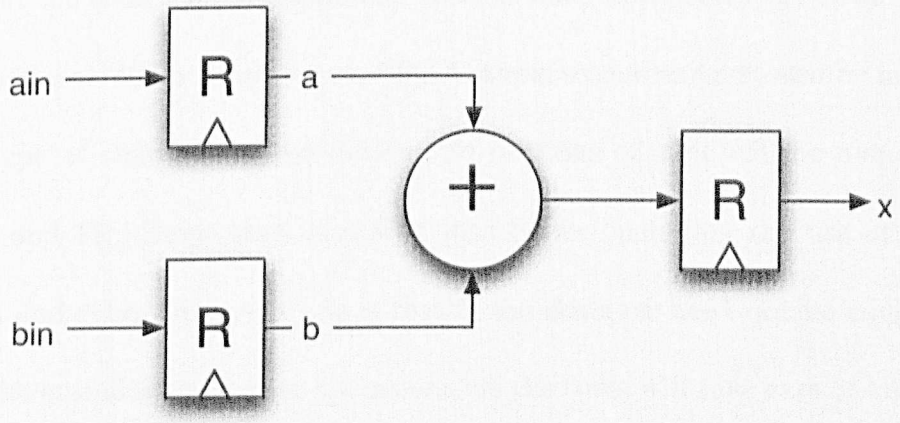


Figure 2.3: Circuit corresponding to the VHDL and Handel-C descriptions in Figures 2.1 and 2.2.

the block under test, and feeds it the appropriate data, while monitoring the block's outputs. Tools such as Modeltech's ModelSim [Mod] can be used to analyse waveforms of the outputs as well as signals internal to the block. It is necessary to consider "corner cases", or patterns of data, perhaps unexpected, that could expose some frailty in the system, for example, overflows in arithmetic operations. Once functional verification is complete, the designer's work is for the most part done.

Synthesis and Mapping

When designing using RTL or behavioural descriptions, these must be translated into primitive hardware blocks. This job is done by the synthesis software, for example Synplicity Synplify Pro [Syn]. An addition specified on a line of code will be turned into an adder. Registers will be created to hold values and the code will be translated into the hardware primitives of the target architecture. When targeting an FPGA, these resources must also be mapped to a type of resource on the device. So portions of logic will be mapped to LUTs and Flip-Flops then clustered into Slices, including the use of carry-chains and other resources. As a result, the designer need not be concerned with the granularity of most operations, as the tools will take care of breaking down large blocks into the size required by target hardware. The output of this stage is a netlist describing hardware resources and the connections between them.

For heterogeneous architectures, the mapping phase is also where decisions

are taken as to which type of resource to use. Often, there will be more than one way to implement a hardware block. For example a multiplier could be implemented using standard logic elements or using an embedded multiplier, a FIFO could be implemented using Flip-Flops or embedded memory. Many synthesis tools allow the designer to specify some preference that is taken into account when synthesising. In other cases, the designer must specify the type of resource to use explicitly. There exists a significant body of research that looks at the trade-offs involved in selecting different types of hardware, and the resultant choices that can be made when designing a reconfigurable architecture. [Smi07] contains a good survey of work in the field.

Placement and Routing

Once the netlist has been produced and the whole circuit has been mapped to the primitives available on the target device, it is necessary to place these instances into specific locations on the device and route the associated signals. The primary aim is to minimise delays by placing communicating blocks adjacent to each other. Once the blocks are placed, it is necessary to configure the routing between them. Again, this is done with the aim of reducing delays. The result of this step is a bitfile, containing all the configuration data needed to set up the circuit in the FPGA device. Typically, in a production system, this bitfile is stored in a Programmable ROM (PROM), which is accessed by the FPGA on power-up, and used to configure the device.

Timing Verification

After placement and routing, it is necessary to check that specified timing constraints have been met. This is often reported when placement and routing is completed. It is also possible to configure the tools to produce a post-place and route simulation model which can be used to check the resultant circuit using tools like ModelSim. If timing constraints have not been met, the violating paths will be reported, and the designer can attempt to modify those parts of the circuit in order to increase performance.

2.2.7 Circuit Measurement Metrics

When designing a circuit, it is necessary to have metrics by which the design can be evaluated. The most obvious metric is that of speed. A faster circuit will process data at a quicker rate. It is important, however to bear in mind that speed should be measured in terms of throughput and not necessarily cycle time. It is possible for a circuit to be clocked at very high speed, but have a relatively low throughput when compared to another circuit with lower clock speed. This is clear when comparing an FPGA implementation against a standard GPP. As shown in Chapter 4 an Intel Pentium 4 Processor running at 2GHz can be outperformed by an FPGA running at 80MHz because the throughput is higher due to parallelism and a more tailored datapath. In image- and video-processing applications, throughput is usually given in frames per second (fps), though it is important to consider the size of each frame too.

In some instances the throughput requirements of a system are fixed, and hence as long as these requirement constraints are met, it is area and power that are of interest.

When discussing the area usage of an FPGA design, the most basic element to be considered is the standard logic element of the target device. In the case of Xilinx FPGAs, area is typically measured in Slices, while for Altera FPGAs, the Adaptive Logic Modules (ALMs) for more recent devices or the Logic Elements (LEs) for older devices, are the measure. Synthesis tools will often report finer levels of resource usage such as flip flops and LUTs, since often the resources can be assigned and packed differently.

With heterogeneous architectures it is also necessary to consider the usage of other resources such as embedded memories and multipliers, since the portions of the circuit mapped to these resources are not represented in the standard Slice count. It is important to note that the proportions of different resources in a device are fixed by the device vendor. As such it may well be possible to deplete the available resources of one type while still having capacity to spare in another type of resource. In such cases, the designer can either target a larger device, or map parts of the circuit from one resource type to another, bearing in mind the performance costs. Memories and multipliers can easily be mapped to use LUTs if needed. Similarly, if some of the standard logic elements are in short supply, the designer should ensure that all portions of the circuit that can be mapped to other resources are transferred.

Another metric that has more recently gained popularity is that of power.

ASIC designers will typically have more control over this, as they can design their architecture to suit. With FPGAs, the power consumption is often out of the hands of the designer, as the static power consumption of the FPGA – which is a subject of the FPGA architecture and manufacturing process – is significant when compared to the dynamic power – caused by circuit switching, and over which the designer may have some influence. In all, FPGAs still consume a significantly greater amount of power than ASICs, at present, and so fail to be considered as an ideal platform for mobile devices. With each transition to newer circuit technologies, the ratio of dynamic power to static power consumption increases, meaning that power considerations will become important to designers in the near future. Currently, the tools provided for power analysis are limited, and techniques for optimising for power obscure. FPGA vendors are pushing forward with architectural solutions such as switching off parts of the chip not in use [TT07]. Development on the two fronts will be necessary in order to assist designers in optimising their designs for power consumption.

When synthesising a design, it is possible to increase the effort level of the tools. This may be necessary when a timing constraint is missed by a small margin, or the design is slightly too large to map to the target device. When the effort is increased, the tools run more aggressive optimisation routines in order to meet constraints. The tools can be instructed to optimise for area or speed, depending on the requirements of the design.

2.3 Hardware Acceleration of Vision Systems

FPGAs provide an ideal platform for prototyping computer vision applications. Such applications are characterised by the large amounts of data processed, and the high computational complexity of the algorithms involved. The provision of heterogeneous resources fairly recently, makes them an even more attractive target platform. Memories specifically can simplify designs hence requiring fewer accesses to external memory, which can often be a bottleneck in a video- or image-processing design. In this section, an overview of some FPGA implementations of vision systems will be presented.

2.3.1 Colour to Black and White Conversion

Images in digital systems are usually represented in the additive three-colour Red-Green-Blue (RGB) domain. However it is often more intuitive to use the HSI (Hue, Saturation, Intensity) colour space. The hue is the colour that is being represented, saturation is how pure the colour is, with full saturation being very strong and zero saturation being grey. Intensity is the brightness of the pixel.

The human vision system is more sensitive to certain wavelengths of light than others, hence in converting from colour to black and white, the colours need to be mixed in different proportions. The standard proportions are shown

in (2.1), where Y is the (greyscale) intensity value [Rus02].

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{2.1}$$

For video applications, the input will often be in colour, but most of the complex computation can be performed on the greyscale intensity image to lessen the computational load, then later combined with the colour image if needed. Here, a simple implementation will be presented, to show how for each algorithm, thinking in terms of the target platform is paramount in an efficient implementation. To implement this colour to black and white conversion efficiently, one can recall that in hardware, any base 2 divisions are simply shifts of the bits, or selecting certain bits. Hence, it is possible to come close to the equation mentioned above without having to implement any multipliers. An example is shown in (2.3), where $\tilde{Y}$ is an approximation of the pixel intensity, given R, G and B input values for red, green and blue respectively. This is much more easily implemented in hardware, with only shifts and 4 two-input adders, as shown in Figure 2.4.

$$\tilde{Y} = \quad 0.3125 \cdot R + 0.5625 \cdot G + 0.125 \cdot B \tag{2.2}$$

$$= \quad \left(\tfrac{1}{4} + \tfrac{1}{16}\right) \cdot R + \left(\tfrac{1}{2} + \tfrac{1}{16}\right) \cdot G + \left(\tfrac{1}{8}\right) \cdot B \tag{2.3}$$

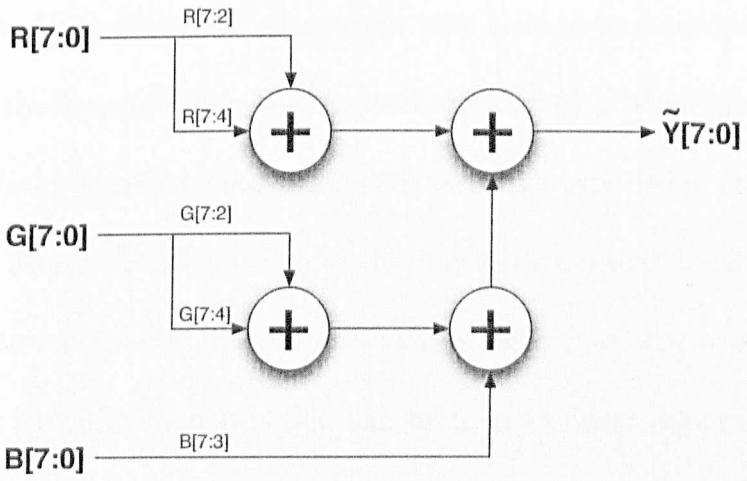


Figure 2.4: Simple colour to greyscale conversion circuit using only adders.

2.3.2 Object Detection

In object detection, a system must detect the presence of an object of interest in the frame. This can be done in one of two ways. Firstly, it is possible to detect the presence of an arbitrary object. This can only be done with a video sequence, since it is the temporal information that provides the cue that something has been introduced into the scene. Such a form of object detection will often be done using background subtraction, where subsequent frames are extracted from a static reference frame, and areas where there is a difference are treated as objects. Unfortunately, this method requires the camera to be stationary and can be adversely affected by other changes such as illumination. Another method is to use motion vectors to extract moving objects in a video sequence. Areas of the image where the motion vectors close to one another move in the same direction and with the same magnitude are considered as rigid bodies, and can be segmented from the frame. It is important to note that some forms of motion yield different patterns of motion vectors, and depending

on the application, alternative clusterings may have to be considered [YJS06]. With both these approaches, this segmentation gives a bounding box which must then be processed to see whether it is of the type being considered in the system. Simple heuristics such as the consideration that a walking person could only be represented by an object that is taller than it is wide, or that a vehicle may be wider than it is tall, can be used to reject segmented objects without having to process them.

The second type of object segmentation involves analysing the frames with some criteria in mind. Such techniques can depend upon colour, texture, shape or other properties of the image. Often the segmentation can be applied to a single frame, with no need for temporal information. In such a system, objects of interest are often directly segmented from the image.

2.3.3 Object Segmentation

Following detection of the presence of an object, it is often necessary to segment this object from the frame. In many cases, detection and segmentation occur together, such as some background subtraction, and most face authentication algorithms. If this is not the case then a number of methods can be employed. Sometimes detection will return the centroid of the desired object, or some other general positional information. In such cases, prior knowledge of the shape of the object can be used to segment it based on these parameters.

Segmentation can also be achieved through the use of edge detection, whether on the image directly, or some property obtained from the detection

phase. Edge detection is most often achieved through the use of simple filters like the Sobel or Laplace masks [HS92]. The result of filtering is an edge map where edge pixels have high values and non-edges have close to zero values, or in the case of the Laplace mask, the edges are represented by zero-crossings. Typically, these edges maps correspond to the outline of objects in the scene. Further shape analysis methods can be used to segment the desired objects.

2.3.4 Object Tracking

In the case of video sequences, temporal data can be used to track the positions of moving objects in the frame. Numerous methods exist (see [YJS06] for a through survey). Typically, motion vectors will be extracted for a frame. Where a cluster of pixels are determined to have the same motion vectors, a rigid body can be deduced. It is important to note that changes in the background, or camera position introduce many superfluous motion vectors. In some systems, the motion vectors are computed for image features rather than directly on pixel blocks. This reduces the amount of data that needs to be processed.

Block matching is achieved by looking at two successive frames and attempting to find the correspondence in one frame for a block in another. The search is typically constrained within a window. Hardware implementations of block matching algorithms are numerous. The most basic block-matching technique is what is termed Full Search Block-Matching (FSBM); in which the correspondence for every block within the search area is calculated in order to

find a match. Clearly this is very computationally expensive. Parallelism can be used to speed it up in hardware, but memory bandwidth and data organisation present challenges in designing an architecture. Simplifications have been suggested including the 3-step search (3SS) [KIH+81], the New 3-step search (N3SS) [LZL94] and the 4-step search (4SS) [PM96]. These are much less computationally intensive, but the search can get stuck in local minima and so these algorithms do not perform as well as FSBM in terms of the quality of results. The FSBM can be implemented in a number of different ways including systolic array designs [KP90, YH95], and tree structures [LCTW97]. A genetic algorithm has also been applied to block-matching [LW96], implemented in [WS03]; the quality of results is much better than for other hierarchical searches with comparable hardware usage, but still a significant saving over FSBM. [TCJ02] includes a good overview of the data reuse and bandwidth needs of different implementations in hardware.

Aside from these direct feature matching methods, other higher-level tracking algorithms include Kalman filters [Sor85], the CONDENSATION algorithm [IB98] and particle filters [LZP03].

2.3.5 Literature Summary

A concise summary of some computer vision algorithms is presented here. Unfortunately, much of the published literature does not follow a systematic approach when reporting hardware designs. In many papers, resource usage is ignored or reported in a non-detailed way. In some cases figures for clock-speed

are given without discussing throughput. In many cases, no mention is made of the heterogeneous resources used in the design. Some implementations simply map software code to hardware through the use of high-level languages, and as such the performance is not accelerated by any significant amount. The aim of this section is simply to give a general idea as to the recent work implemented on FPGAs, with a specific concentration on face detection.

In [RV04], the authors develop a hardware implementation of an edge-detection system based on the Canny edge detector. This algorithm first smoothes the image, then computes the horizontal and vertical gradients for each pixel using the standard Prewitt kernels. The result is that edges become ridges. Using non-maximum suppression, all non-edges can be eliminated. Finally, the resultant image is thresholded to select the significant edges. The hardware system was designed using Handel-C and resulted in a system that ran at 16MHz on a Xilinx Virtex E, processing a 256×256 pixel image in 4.2ms.

In [HWCC05, HCWC06], a similar algorithm is implemented, inspired by an implementation in [AC97]. Firstly, instead of applying the Gaussian smoothing filter as is, it is approximated using fractions that can be represented using powers of 2. This simplifies the filtering circuitry significantly, and this is implemented using a systolic array. An edge-strength and localisation unit is developed which computes edge values and compares them with neighbours in the edge direction producing a '1' for an edge pixel in the output. Two different FIFO schemes are compared to allow flexibility in image size (while maintaining a dimension that is a power of 2 as a requirement).

The end result is a system that runs at 73.6MHz with a throughput 265 times greater than a DSP implementation.

In [SV07], the authors apply simple edge detection using standard Prewitt masks, then by summing pixels horizontally and vertically, extract the grid array structure for DNA microarrays. They use a Xilinx Virtex II Pro FPGA, employing the PowerPC processor to manage their hardware system and Block RAMs to buffer parts of the large input images as they are processed, and similarly at output. A Block RAM is used for storing the horizontal and vertical profiles. The whole system is clocked at 200MHz and shows an order of magnitude improvement over software in terms of processing speed. Input images vary in size from around 1900× 1900 up to around 2000× 5600 pixels.

In [PPAC06], the authors develop an FPGA-based system that can detect vehicles in aerial images for use in an unmanned aerial vehicle (UAV) system. The system operates on streaming data from a camera, due to memory constraints. First the pixels are converted to the HSV (Hue, Saturation, Value) colour space. The pixels are then thresholded independent of hue, and the resultant image eroded to remove superfluous points. Edge detection is then applied using a simple Laplacian kernel. The resultant image contains blobs which are then correlated with a template that takes into account the expected size and spacing of the desired objects, in this case vehicles. The system is shown to perform as desired though no performance figures are given.

In [DDM+05], the authors develop a system to check the bolts in railway lines. They first predict the frames in which bolts are expected using the dis-

tance between bolts; this is done in software. The images are then processed using the Haar DWT [SN96] transform and a Neural Classifier used to determine the presence of bolts. The combined software-hardware system processes a 24×100 pixel window in 13.29 μs.

In [APTE+05], a system is presented that implements Wronskian change detection on video sequences. A reference frame is subtracted from subsequent frames with some measure of robustness to noise and illumination changes. The system uses external memories on a development board for storing the reference and result frames. The system occupies just over 7000 slices in a Xilinx Virtex FPGA clocked at 25MHz, processing 640×480 pixel frames at 15fps.

In [JSC06], the authors develop a Hough transform system for detecting circles in images. An image is first converted from colour to greyscale simply by averaging the channel values. Then a Sobel edge detector is applied followed by a Laplace filter to reduce the number of edge pixels further. A simple unit based on the Circular Hough Transform (CHT) is implemented, producing a histogram of circle parameters, from which the maximum is taken to be the dominant circle in the image. The design was implemented on an Altera Stratix FPGA, using 3056 Logic Cells, 128 M4K RAMs, 2 MRAMs and 42 DSP blocks. The design is not significantly pipelined though, so despite being clocked at 50MHz, it takes 4.3 seconds to detect the circles in a 256×256 pixel image.

The task of detecting people in images is one of the more interesting areas

in vision research. One of the simple methods involved in many such systems is skin detection based on colour. In [TMJF06], an FPGA system for the detection of skin colour is presented. They combine classifications obtained from the YIQ, YUV, YCbCr and RGB colour spaces since each performs well in specific instances. Colour space conversion was implemented using the Xilinx System Generator tool with fixed coefficient multiplications. System Generator was also used to implement the rule checks. A probability map was implemented in Block RAMs with a MicroBlaze soft processor used to tweak parameters in the various blocks. The system processes 640×480 pixel images at 190 frames per second.

In [Ooi06], a similar colour segmentation routine is implemented, but it is done by translating C code into a Handel-C implementation and using the Celoxica PAL API which allows simplified access to board peripherals [Cel]. While the hardware system is shown to perform close to the software implementation in terms of accuracy, no performance figures are given, and the area usage is given in NAND gates, though how the numbers are obtained is not clear.

[RAE04] details a rather unusual approach to hardware design. Following a similar algorithm to the previous paper, the authors implement a custom MIPS processor to conduct the face detection by analysing the colour values of pixels. The system is also implemented using Handel-C and the Celoxica PAL libraries. The system processes 360×288 pixel frames at 13 frames per second. It is unclear why the authors did not implement the algorithm directly

in hardware, as clearly the performance is below-par and the effort needed to design a processor may well be as significant as designing custom hardware.

The implementation in [PB03] is also colour-based. The system takes a 176×144 pixel input frame and applies 16×16 pixel subsampling resulting in an 11×9 frame. They claim that this is done to reduce computational complexity as well as generating larger skin patches. Pixels are mapped to the LogRB colour space then compared to a histogram of skin values obtained during a training phase. The detected skin areas are then enhanced through spatial filtering, and the centroids extracted using a moments based calculation. Subsampling is done on the image stream directly, forgoing the need for buffering. The colour space histogram is stored in embedded memory and used to apply the skin filtering operation. The system is designed to adapt the values in the histogram based on detected faces. The circuit is small, as would be expected from the data size being processed. It is clocked at 33MHz and occupies approximately 3000 Logic Elements in an Altera FLEX 20K FPGA processing 434 frames per second.

In [WBC04], the authors present a hardware implementation of the AdaBoost algorithm which has been shown to offer excellent performance for face detection [VJ01]. The system combines several weak classifiers based on Haar wavelets into a strong classifier. A pyramid of sub-images is created, so from one 120×120 pixel image, 17,281 sub-images must be processed. An integral image is computed using Multiply-accumulators (MACs) for each 24×24 pixel sub-image. Three classifier stages with an increasing number of Haar wavelets

are implemented. The resultant circuit is clocked at 91MHz and occupies 8000 Slices, 56 Block RAMs and 28 embedded multipliers in a Xilinx Virtex II 2000 FPGA. It processes 120×120 pixel frames at 15 frames per second.

In [ITNI06], the authors develop a face detection system based on Neural Networks [RBK98]. A shared MAC is used between different neurons, each with their own storage. Different scales of image are processed using a 20× 20 pixel window to allow for different sized faces. Little detail of the implementation is given, but the system is clocked at 100MHz and processes 320×240 pixel image frames at 40fps.

In [NHAS06], a face detection system is developed based on a Naive Bayes classifier. Again, the image is processed in 20×20 search windows. The windows are made to overlap in order to provide full coverage. First Sobel edge detection is applied to the images after they have been histogram-equalised. Then each window is classified as a face or not-face using the Bayes classifier which has been trained on a large training set. A face is located by looking at clusters of windows that have been identified as faces. The position of the lips is identified by analysing the lower portion of the face segment. The points of highest contrast are the left and right edges of the lips. By tracing along the lip edge, the upper and lower sections can be identified. The system was implemented on an Altera Stratix FPGA clocked at 41MHz processing 136 320×240 pixel frames per second. The overall architecture is shown in Figure 2.5.

In all of these implementations, it is clear that those that involved the de-

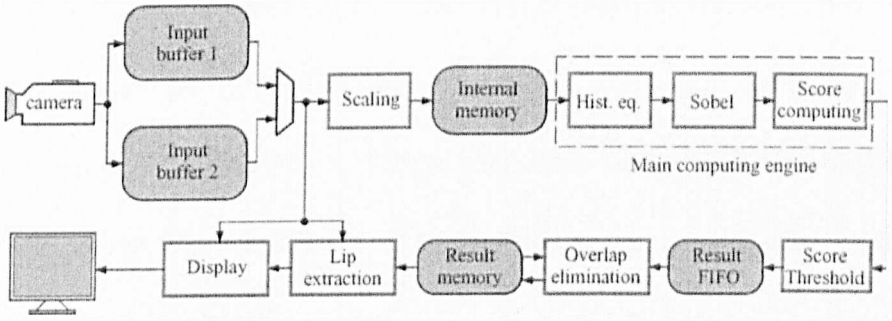


Figure 2.5: The system architecture for the face detection and lip extraction implementation in [NHAS06]. © 2006 IEEE.

sign of a custom architecture with significant pipelining and the use of embedded elements, offered a significant performance advantage over software. Those implementations in which the hardware seemed an afterthought, achieved little benefit over software. It is thus clear that to exploit the power of hardware, an implementation must be deigned in a way that suits the target platform.

2.4 Summary

It is clear that FPGAs, particularly modern heterogeneous FPGAs, are ideally suited to the field of computer vision. The high computational complexity and significant data richness of vision applications means that software implementations are often insufficient for the real-time performance usually demanded. Given the plethora of algorithms for any given task, as well as the significant variation in parameters and other design aspects, the flexibility of FPGAs provides an ideal platform for investigating and tuning vision systems. While the expertise requirement is more than for software systems, an FGPA designer's

work is significantly simpler than of an ASIC designer, and can reap benefits quicker and more cheaply. The most important conclusion from this chapter is that considered, appropriate design decisions, taken with the target platform in mind are what mark the dividing line between successful, significant acceleration and a mediocre showing.

Chapter 3

Trace Transform Theory

3.1 Introduction

The Trace Transform is a novel algorithm, first introduced in 1998 by Kadyrov and Petrou [KP98]. The name belies the fact that the Trace Transform does in fact describe a class of algorithms rather than a specific case. Essentially, the Trace transform of an image is constructed by computing some functional along all lines crossing the image. The specific functional is not pre-determined, but rather selected to suit the application. In order to better understand the transform, it is beneficial to look at a more specific case of the transform that was introduced in 1917, namely, the Radon Transform. The Radon transform has garnered widespread use in fields as divergent as Computer Tomography (CT), astrophysics, electron microscopy and nuclear magnetic resonance [Dea83].

The Trace transform can be considered as a generalisation of the Radon

transform, as will become apparent. It has been shown to be a powerful tool in image recognition and categorisation tasks [KP01], though the issue of high computational complexity has been an obstacle to widespread adoption. By investigating the principles behind the transform and understanding the complexity issues, it is possible to identify algorithmic parallelism and thus methods by which this class of transform can be accelerated in hardware. Furthermore, the flexibility afforded by modern Field Programmable Gate Arrays (FPGAs) is ideally suited to such an algorithm, that is itself highly flexible.

3.2 The Radon Transform

Before discussing the detailed theory behind the Trace transform, it is worth introducing its precursor, the Radon transform. The Radon Transform has come to the fore in recent decades primarily as a mechanism for dealing with the reconstruction problem. This is the problem of determining the internal properties of an object through external probing. Essentially, the object either emits or is acted on by a probe; by taking the resultant profile, some internal property of the object can be identified.

3.2.1 Mathematical Foundations

A detailed definition of the transform and thorough explanation of all its mathematical properties can be found in [Dea83]. A basic explanation is reproduced here for reference.

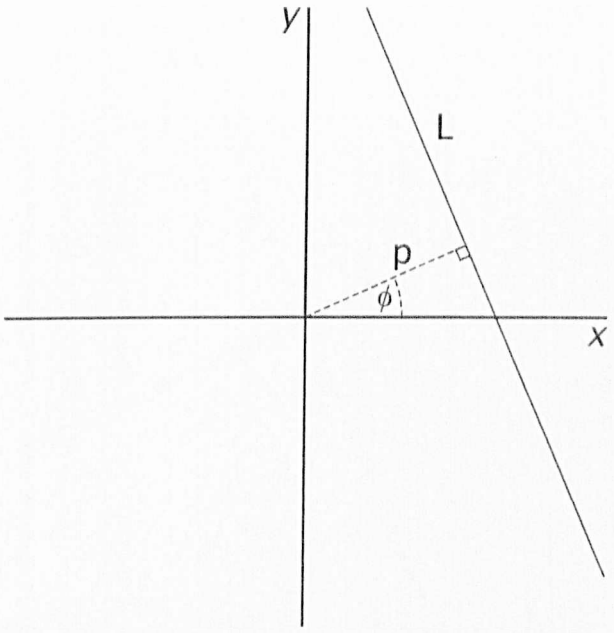


Figure 3.1: Coordinates describing a line L.

Consider an arbitrary function f of coordinates (x, y) defined on a plane of real numbers. If L is any line in the plane, then the Radon Transform (designated by $\mathscr{R}$) of $f(x, y)$ is equal to the mapping defined by the line integral of f along all possible lines L. Explicitly,

$$\check{f} = \mathscr{R}\{f\} = \int_L f(x, y)ds, \tag{3.1}$$

where ds is an infinitesimal increment of length along L. Figure 3.1 shows a line L with equation

$$p = x\cos\phi + y\sin\phi. \tag{3.2}$$

The line integral, as defined in 3.1, clearly depends on the values of p and

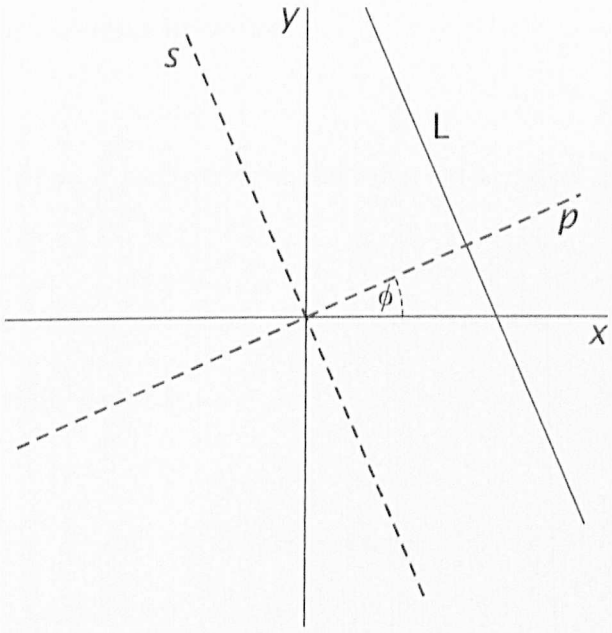


Figure 3.2: Coordinates describing a line L relative to original and rotated coordinates.

ϕ. This can be stated explicitly as follows:

$$\check{f}(p,\phi) = \mathscr{R}\{f\} = \int_L f(x,y)ds. \tag{3.3}$$

If $\check{f}(p,\phi)$ is known for all values of p and ϕ, then $\check{f}(p,\phi)$ is the Radon Transform of $f(x,y)$, otherwise it is considered a sample of the Radon Transform.

Consider now, a new coordinate system, introduced by rotating the axes in Figure 3.1 by ϕ, as shown in Figure 3.2. If the new axes are labelled p and s, then

$$x = p\cos\phi - s\sin\phi$$

$$y = p\sin\phi + s\cos\phi$$

The transform can then be written

$$\check{f}(p,\phi) = \int_{-\infty}^{\infty} f(p\cos\phi - s\sin\phi, p\sin\phi + s\cos\phi)ds. \tag{3.4}$$

Another common way of writing the Radon transform is in vector notation. First, define the unit vector

$$\xi = (\cos\phi, \sin\phi). \tag{3.5}$$

Now, the equation of the line can be written as

$$p = \xi \cdot \mathbf{x} = x\cos\phi + y\sin\phi. \tag{3.6}$$

and the Radon transform can be written

$$\check{f}(p,\xi) = \int f(\mathbf{x})\delta(p - \xi\cdot\mathbf{x})d\mathbf{x}. \tag{3.7}$$

The Radon transform is linear and homogeneous. It is also closely related to the Fourier transform [Dea83].

Key to the Radon transform's use is the inversion property: that is, the ability to deduce the original function from the transformed profiles. Backprojection is the preferred method for achieving this inversion and is stated here for reference.

Consider an arbitrary function $\psi(t,\xi)$ where $t = \xi\cdot\mathbf{x} = x\cos\phi + y\sin\phi$.

The backprojection operator $\mathscr{B}$ is defined by

$$\mathscr{B}\psi = \int_0^{\pi} \psi(x\cos\phi + y\sin\phi, \xi)d\phi. \tag{3.8}$$

This can also be written

$$[\mathscr{B}\psi](x,y) = \int_0^{\pi} \psi(x\cos\phi + y\sin\phi, \xi)d\phi, \tag{3.9}$$

since $\mathscr{B}\psi$ is a function of (x,y) and ξ completely depends on ϕ.

Now, if $\psi(p,\phi)$ is set to the projection function $\check{f}(p,\phi)$ obtained by applying the Radon Transform to $f(x,y)$, then the contribution to $\mathscr{B}\psi$ at point (x,y) is just $\check{f}(p,\phi)\cdot d\phi$ for any given ϕ. The value of $\check{f}(p,\phi)$ is simply the integral of the line passing through (x,y) at distance $p = x\cos\phi + y\sin\phi$ from the origin. Integrating for all ϕ yields the complete backprojection as per (3.9). This is shown in Figure 3.3.

3.2.2 Applications

To better understand how the Radon Transform is used in practice, it is worth considering the many applications that make use of it. As mentioned previously, the transform facilitates the extraction of information about the internal structure of an object of interest from a set of profiles. These profiles are formed from some application of a probe to the object and taking measurements along various lines as defined previously, thus constructing the Radon

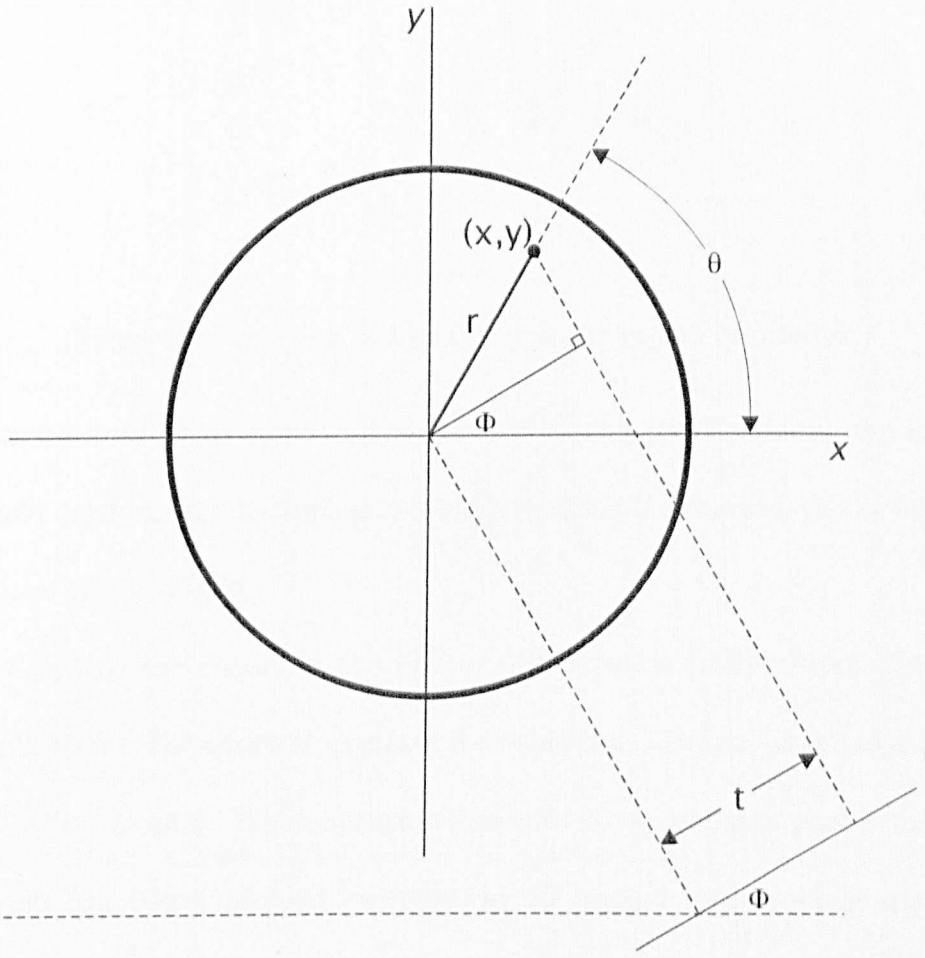


Figure 3.3: Geometry for obtaining the ϕ-backprojection. For a fixed angle Φ, the incremental contribution $d(\mathscr{B}\psi)$ to $\mathscr{B}\psi$ at the point (x, y), or alternatively (r, θ), is given by $\psi(t, \Phi)d\theta$. The full contribution to $\mathscr{B}\psi$ at (x, y) is found by integrating over ϕ as shown in (3.9). Note that $t = x\cos\Phi + y\sin\Phi = r\cos(\theta - \Phi)$. Adapted from [Dea83].

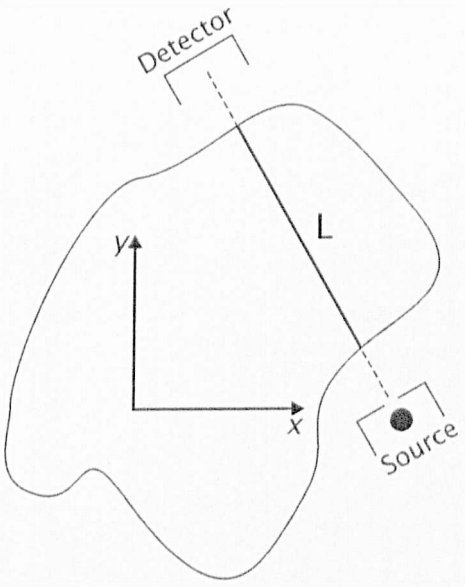


Figure 3.4: A beam passes through the region of interest.

Transform result from these (p, ϕ) values. With this result in hand, the Inverse Radon Transform can be applied to obtain detailed information on the internal structure of the object.

As regards the nature of the probe, this depends entirely upon the field of application. The simplest example to consider is X-Ray Computed Tomography (CT) [Dea83]. When a narrow beam of X-ray photons passes through an object, the beam intensity decreases by an amount that depends upon the density and nuclear composition of the materials in its path. When a single cross-section of the object is considered, detecting the amount that has passed though gives a single projection, equivalent to the line projection defined in 3.1. Multiple parallel projections would yield a single profile $P(p, \phi)$ for a single value of ϕ, as shown in Figure 3.5. By applying the same technique at further angles, a complete sample of the Radon transform is obtained. The transmitted amount of radiation is determined by known equations. This can

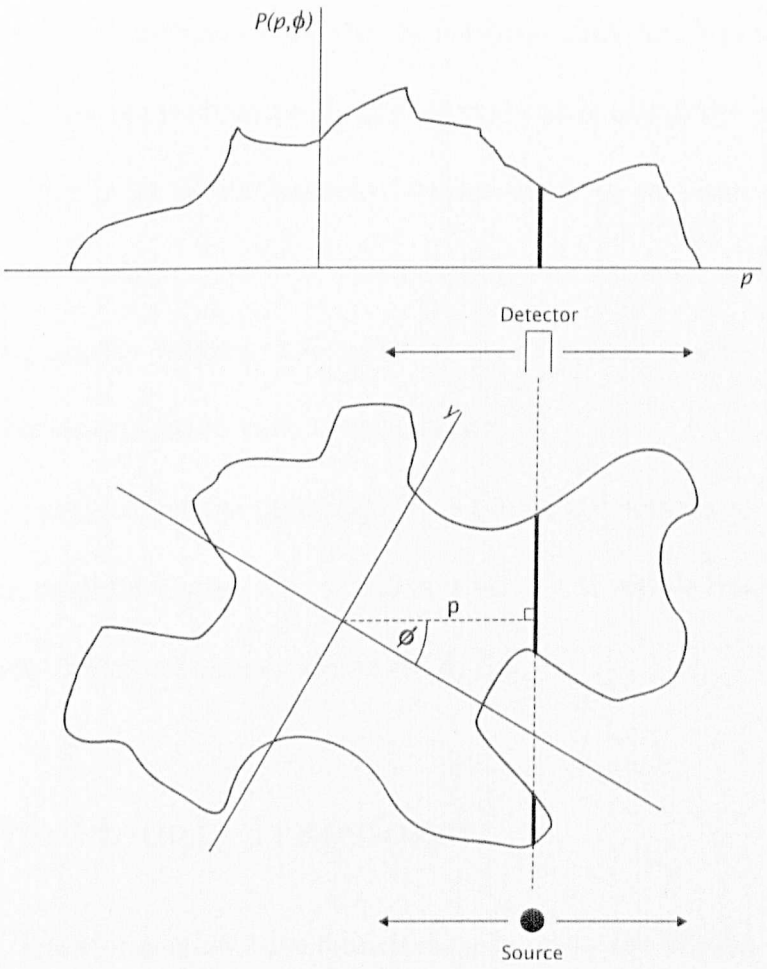


Figure 3.5: For a fixed angle ϕ, the source and detector move (varying p) and this creates a profile, $P(p, \phi)$ for angle ϕ. (Adapted from [Dea83]).

then be inverted to give a full picture of the composition of that cross-section.

3.2.3 Application to Image Processing

Applying the Radon transform to the image domain is a trivial development. Replacing the beam and probe, one simply considers the sum of pixel intensities along the lines that cross the image. It is important to note that digital images being discrete in nature means that some method for approximating the values along lines is necessary, since often the line will not pass directly through the

centre of a pixel. There are a number of methods that can be used. Firstly, nearest-neighbour approximation simply takes the intensity of the pixel nearest to the sampling point as its intensity. Other methods of interpolation aim to give a more accurate result by interpolating the intensity values of other neighbouring pixels. Bilinear interpolation uses the four nearest neighbours while bi-cubic interpolation uses 16 neighbours.

To better understand the mappings from the image domain to the parameter domain, consider Figure 3.6. It shows a variety of simple images (a, c, e, g, i) and their transformed equivalents (b, d, f, h, j).

3.3 The Hough Transform

The Hough Transform is another transform related to the Radon transform. The idea behind the Hough transform is to characterise the shapes in the edge-map of an image. The Hough transform maps a line in the image domain to a single point in the Hough domain. The Hough domain parameters are the same as those of the Radon and Trace transforms. In fact, the application of the Radon transform to an edge map would yield the Hough transform. The difference can be thought of theoretically as follows: the Radon transform maps a point in the image domain to a shape in the parameter domain, whereas the Hough transform maps a shape in the image domain to a point in the parameter domain. To clarify this, it is suggested that each point in the image domain "votes" for the parameters that correspond to all the lines that

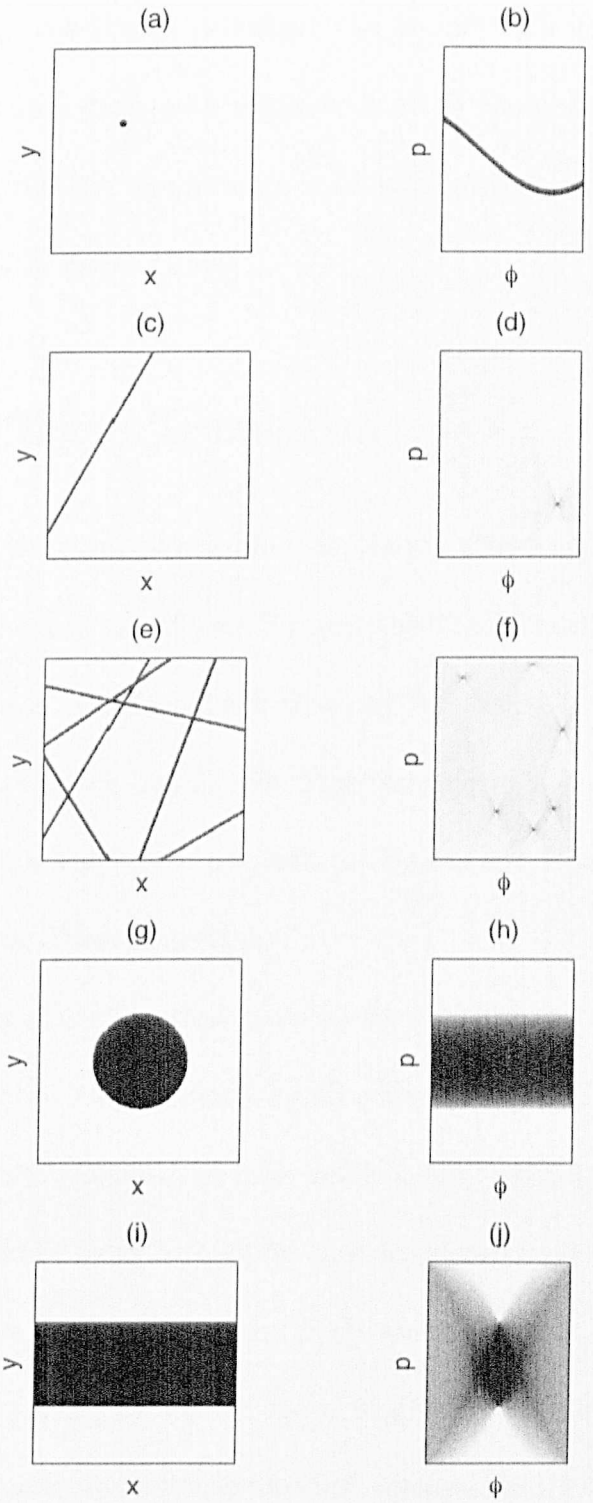


Figure 3.6: Some basic images and their equivalents in the Radon parameter domain.

pass through it. While the theory is different, the manifestation is the same. Computationally, the Hough transform can be more efficient than the Radon transform, as it only deals with edges, so most of the pixels to be considered are zero-valued. If taken into account in an implementation, this can make the system significantly faster [Tof96].

3.4 The Trace Transform

After discussing the principles behind the Radon transform, it is now possible to introduce the Trace transform. Simply, the Trace transform is a generalisation of the Radon transform such that any functional can be used in place of the integral along each line L. The Trace transform is thus a class of transforms rather than a single instance, and the Radon and Hough transforms are special cases of the Trace transform.

A functional is a function that maps a vector function to a single value, and requires that function to be defined for all parameter values. With a standard function, the result is defined on each single point, whereas a functional will only produce a result from a function that is defined over all points. As an example $\int f(x)$, $max(f(x))$ and $min(f(x))$ are functionals, since the full set of values for $f(x)$ must be known for a result to be valid. In the Radon transform, the functional is simply the line integral; in the Trace Transform, the functional, denoted by $T(f(t))$, is not specified, and can be any functional of one's choosing. Herein lies the flexibility of the Trace transform and its

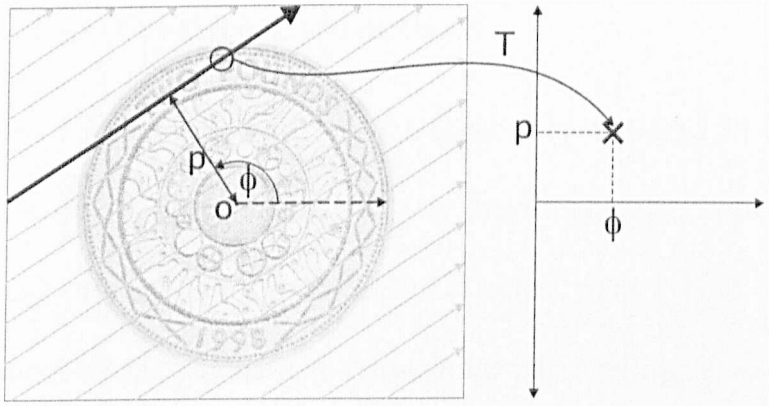


Figure 3.7: Mapping of an image to the Trace parameter domain.

power in recognition tasks. A diagrammatical representation of the domain parameter transformation is shown in Figure 3.7.

The mathematical definition of the Trace transform is identical to that of the Radon transform in terms of parameters. Just as with the Radon transform, the Trace domain is defined by parameters (ϕ, p). Each (ϕ, p) point in the Trace domain is the result of applying a functional, $T(f(t))$, to the line corresponding to parameters ϕ and p.

It is important to note here that the relationship to the Radon transform is only in terms of definitions. The properties and relationships to other transforms do not hold for the Trace transform, since each functional is different. Separate investigations would have to be undertaken to establish the degree of similarity in these properties, but since the Trace transform is not being used for the reconstruction problem, properties such as linearity and inversion are of no concern. It is possible to select a functional that breaks Radon transform properties and yet shows effectiveness when used in a Trace transform application.

3.4.1 Triple Feature Extraction

One of the methods by which the Trace transform is employed for recognition tasks is "Triple Feature" extraction. So far we have observed how the transform maps an image to another two-dimensional space defined over coordinates (ϕ, p). Extraction of triple features requires further functionals to be applied to this result to give single value features. Firstly a functional, called the "diametrical functional" , P, is applied to each column in turn returning a vector in ϕ. Finally, a functional called the "circus functional", Φ, is applied to the vector, returning a single value result called the "triple feature" [KP01]. These steps are shown in Figure 3.8.

Triple features are thus single numbers that characterise the image in some way. At first, this seems like an excessive reduction of an entire image space to a single value. However, the strength of this approach lies in selecting multiple functionals at stages T, P and Φ of the feature extraction process. If N_T functionals are used for the trace functional (T), N_P functionals are used for the diametrical functional (P) and N_Φ functionals are used for the circus functional (Φ), then a total of $N_T N_P N_\Phi$ triple features can be extracted. Taking these triple features together can allow for complex characterisation of a scene.

It is expected that not all of these triple features would play a part in characterising the image, however given sufficient data with regard to the specific application and a wide array of functionals, it is possible to hone a selection

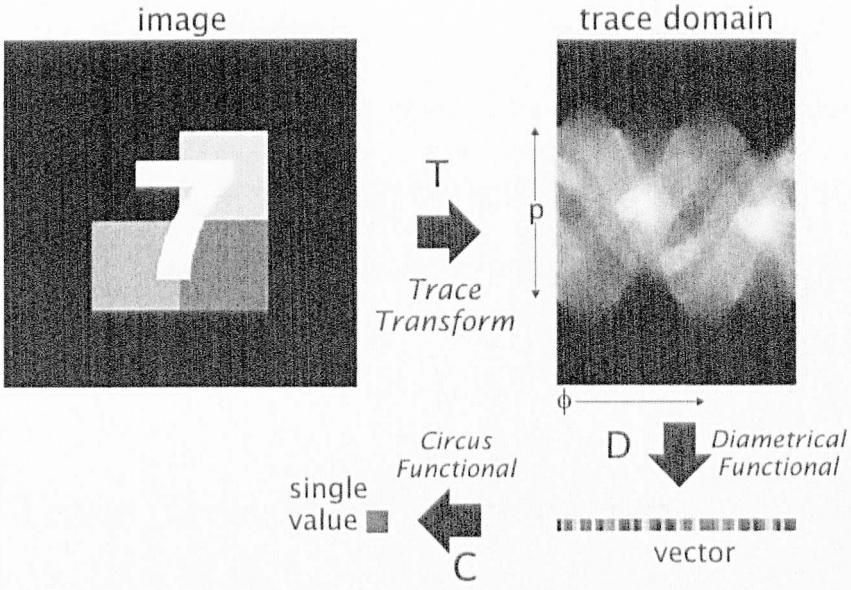


Figure 3.8: An image, its trace, and the subsequent steps of triple feature extraction.

of functionals that effectively characterise the required property for a specific application. In [KP03, PK04], the authors show a system which employs functionals that yield triple features that are robust to affine transformations. In [KP06], features are constructed that are able to deduce the affine transformations that affect the target image.

3.4.2 Selection of Functionals

When the Trace transform is used for image recognition tasks, the requirement is typically to identify matching objects, subject to some distortions, including translation, rotation, scaling, affine transformations, and even some minor non-linear deformations. To achieve this, appropriate functionals must be selected for T, P and Φ. Petrou and Kadyrov detailed the theory behind selection of these functionals in [PK04]. It is clear, intuitively, that certain types

of functionals at different stages can yield robustness to certain distortions. Consider a rotation in the original image: this would yield a horizontal shift in the Trace image. Thus a suitable function selected for Φ could cancel this effect on a triple-feature. More complex transformations can also be analysed in this way.

3.5 Trace Transform Applications

The Trace transform, despite its relative novelty, has been applied to a wide range of different image analysis applications. When considering the flexibility inherent in the algorithm, it is clear that it can serve a purpose in many different fields. Since functionals can be selected according to their performance for a particular task, the end result is an effective and accurate system. In this section, various existent applications of the Trace transform are briefly presented.

3.5.1 Image Database Search

The primary application that was used by the proposers of the Trace transform to show it effectiveness was an image database search system [KP01]. An image is selected from a database containing different subjects, then some transformations are applied. This distorted image is then used as the search subject. By applying triple feature extraction to the subject and comparing the results to the features stored for each of the images in the database, an

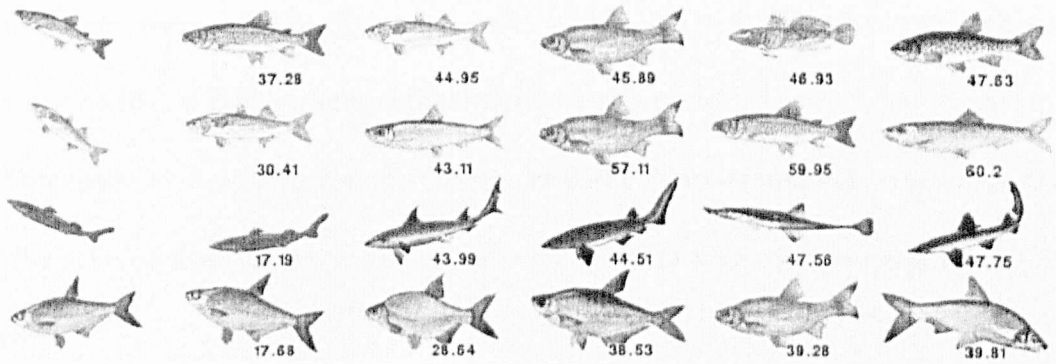


Figure 3.9: Examples of queries to the image database and the first five matches returned for each [KP01]. © 2001 IEEE.

attempt is made to find the correct match. Varying the distortion applied to the subject allows investigation of the algorithm's robustness. Figure 3.9 shows queries to the database that have been distorted by random amounts of translation, rotation and scaling, and the first five returned matches for each.

In [KP01], the authors construct five sets of invariant features for each of 94 images in a database containing pictures of different fish. An image is selected from the database, then distortions such as rotation, scaling, translation and the addition of noise, are applied. This distorted image is then made the subject of a query. The five triple features are extracted for this query image and compared to the values stored in the database. The values that closest match indicate the matching image. Scale factors down to 0.6×, along with random rotation and translation, return the correct image within the first five returned results, the vast majority (88%) being returned as the first match, even in the worst case. Additive Gaussian noise within the object was also shown to be tolerated very well, even in the presence of scaling, rotation and translation. Experiments with "salt and pepper" noise gave similar results.

However, the addition of noise to the whole image (i.e., outside the object outline) had a highly detrimental effect on the performance of the algorithm. This leads to an important conclusion: that the Trace transform, when used for object recognition tasks, must be preceded by some sort of object segmentation step.

In [PK04], this system is extended, and further experiments are conducted in the presence of the full set of affine transformations as well as non-linear deformation and occlusion. A Trace transform system is shown to be immune to affine transformations, outperforming standard moments-based methods. In the presence of noise within the object, the Trace transform becomes far superior to the other methods. The experiments are then carried out with query subjects that have non-linear deformations and illumination changes applied alongside the affine transformation. The Trace transform shows robustness, even in the presence of moderate amounts of these distortions. Furthermore, it degrades gracefully as opposed to the rapid breakdown in the case of the moments-based approaches. Finally, the test is carried out for query subjects that are partially occluded, or have a text label attached. Again, the Trace Transform shows robustness as long as the distortions fall within the outline of the object.

A slight variation is presented in [KP03], where circuses are used instead of triple features; the last step of triple feature extraction is not carried out. The results of applying the first two functional types is a polar plot of the circus functional which is used as an object signature. Matching is achieved

by correlating the signatures of the query subject and the elements in the database. This method is tested for occlusions and found to outperform the standard moments-based approaches.

A number of other image recognition applications along the same lines have also been implemented [TZFO05, STZ05].

3.5.2 Token Verification

In [KP01], the Trace transform is applied to token registration. This task consists of comparing a 2D object to a reference. The test object has undergone some rotation and translation, and these parameters must be recovered in order to allow for an aligned comparison between the two. By selecting sets of T, P and Φ functionals that were sensitive to each of these transformations, a system was developed that could extract the parameters accurately. Even in the presence of noise, the system was robust. Again, noise introduced outside the object outline significantly degraded results.

3.5.3 Change Detection

Also in [KP01], the authors apply the Trace transform to change detection. The idea is that some changes that one seeks to monitor can be thought of a change in texture. The example used is that of a car park, where one may wish to monitor the level of activity, characterised by the number of parked cars. Rather than counting blobs, an aerial photograph of the car park, as shown in Figure 3.10, is regarded as a texture and a large number of triple features are

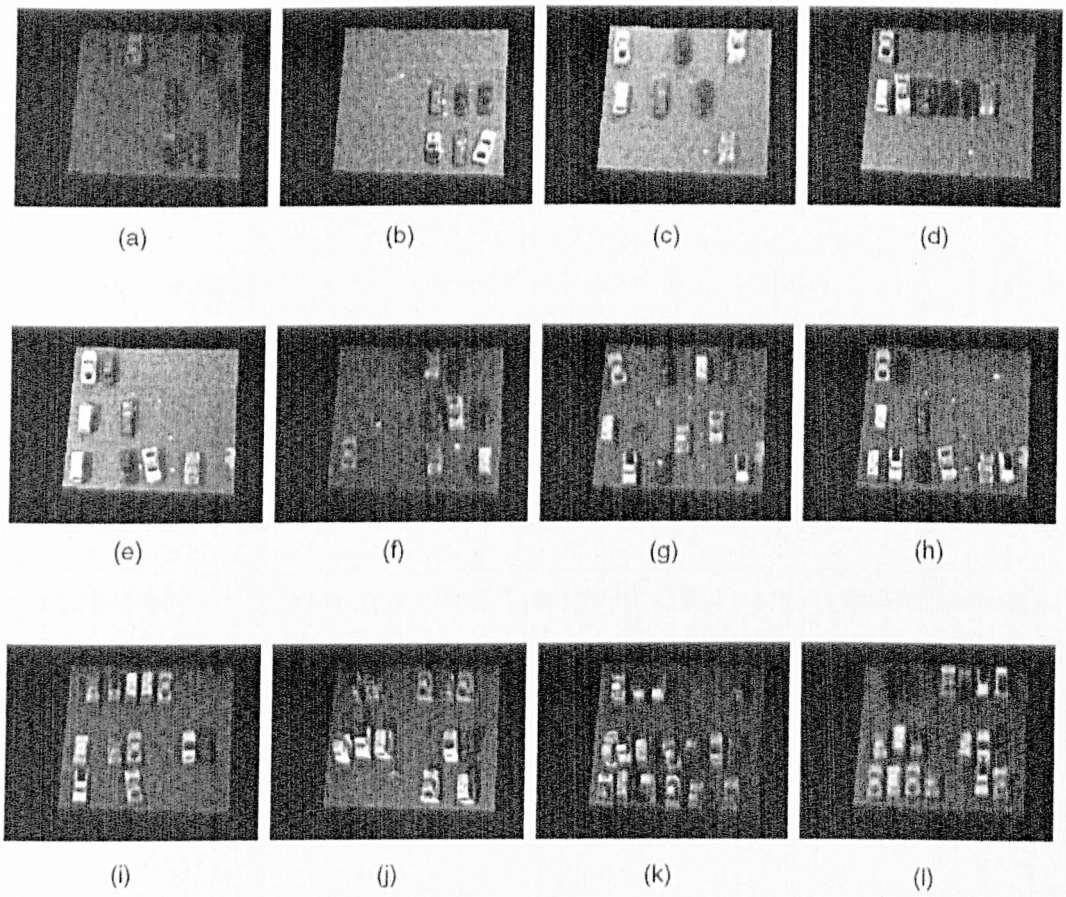


Figure 3.10: Aerial images of a car park with varying degrees of activity. By identifying triple features that correspond to the level of activity, a set that accurately characterises activity can be found[KP01]. © 2001 IEEE.

computed for each of these training images that show the car park in various different states of activity. By finding those triple features that correlate with the level of activity, one can then take any image of the car park, extract the same triple features, and give an accurate conclusion as to the level of activity in the image. In this sense the Trace transform is used to construct a huge number of features, which are then whittled down to the few that, from observation, characterise the required property.

3.5.4 Face Authentication

In [SPKK03, SPKK05], the authors develop a face authentication system using the Trace transform. The principle is to compare a candidate face to a face stored in the database and determine whether it is a match. Such a system must be robust to some changes in illumination, facial hair, expression, and other changes that may occur to a subject's face between two separate occasions. In this implementation, only the first stage of the Trace transform is used, that is the application of the T functional. This yields a two-dimensional Trace image defined over parameters (ϕ, p), as detailed earlier. As with most authentication systems, a training phase is used to derive the recognition parameters which are then used during authentication.

In order to compare the trace image of a candidate face with that of a reference in the database, two separate methods are employed. The first is the Weighted Trace Transform (WTT). In the WTT, training images of the same person are compared and wherever the values in the trace image differ between all training images by less than some threshold, a weight matrix, $W(\phi, p)$ is set to 1. Elsewhere it is set to 0. This matrix defines the points that most characterise this individual face, since they vary the least from one pose to the next. This matrix is used to select pixels in the candidate image to compare to the source image. By looking at the overall level of matching between the two in the positions defined by the W matrix, a confident match can be determined.

The second method used in this implementation is the Shape Trace Transform (STT). Again, trace images are extracted from a candidate image, however instead of matching trace images directly, a shape is extracted from the trace for comparison. This is done by thresholding the image according to some pre-computed threshold then applying edge detection. The resultant shape is then compared to the shapes for faces in the database using a novel measure they propose called the Hausdorff Context based on the Hausdorff Distance [HGR93]. The steps involved in this process are illustrated in Figure 3.11. A match in the shape indicates a match of the corresponding faces. In order to compute the thresholds for both the WTT and STT, reinforcement learning is used, which uses results from previous matches to improve the threshold with each iteration.

The two classifiers mentioned above are combined to create a system that outperforms many other face authentication algorithms. It is worth noting however, that the STT contributes the most to the performance of the system. The WTT only adds a very marginal improvement in performance, and in isolation performs poorly. In [MKS+03], the performance of a system based solely on the STT is compared to seven other face verification algorithms and outperforms them all.

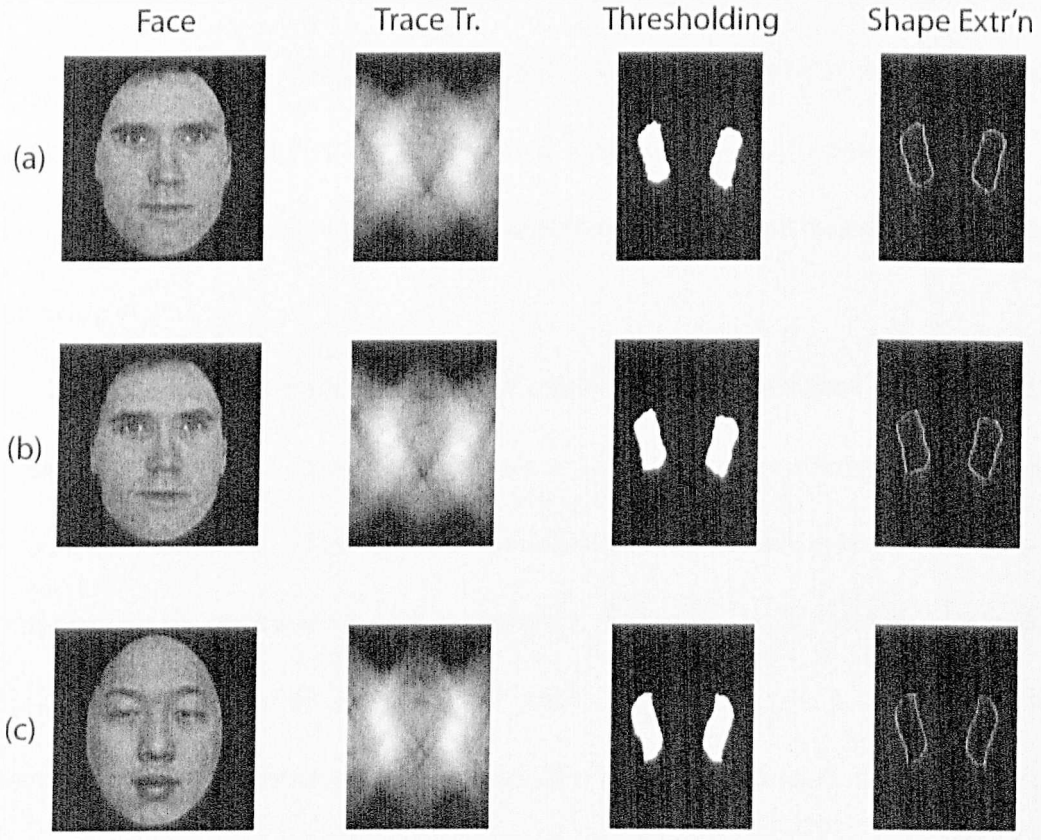


Figure 3.11: The Shape Trace Transform for face authentication. Two faces with different poses, (a) and (b) belonging to the same individual, and another face, (c), of a different individual. The shapes extracted show a match for the same person and a distinct difference for the odd face. Generated using MATLAB as per the system described in [SPKK05].

3.6 Computational Complexity of the Trace Transform

One issue with the Trace transform is that of computational complexity. In this section, a thorough investigation of the algorithm's computational complexity is presented. This analysis is used in Chapter 4 to develop a hardware architecture that implements the Trace transform. To investigate this, it is first necessary to understand the parameters that are controlled in any implementation.

Firstly, a fixed sampling density of angles can be considered, n_ϕ; one could consider angles down to a one degree accuracy or lower, or alternatively choose a coarser sampling. The amount of information carried over to the Trace domain clearly depends on this parameter. Secondly, it is possible to sample an arbitrary number of lines, n_p, per angle. Again this has a bearing on the accuracy of the transformation. For an $N \times N$ image, an inter-line distance of a single pixel may be required, so n_p would have a maximum value of $\sqrt{2}N$ (for a diagonal line). Since each line maps to a single point in the (ϕ, p) domain, an image transformed with the above parameters will yield a trace of size $n_\phi \times n_p$. Another parameter that can be varied is the sampling granularity along each line, or more intuitively, the number of samples taken into account for each line, n_t. If the requirement is to read every pixel along each line, then the maximum value of n_t is also $\sqrt{2}N$. This will not affect the density of data in the parameter domain but will affect the accuracy of the results of applying

Parameter	Explanation
n_ϕ	the number of angles to consider
n_p	the number of distances (inverse of the interline distance)
n_t	the number of points to consider along each trace line
N_T	the number of Trace functionals
N_P	the number of diametrical functionals
N_Φ	the number of circus functionals
C_ϕ	the operations required for trace line address generation
C_T	the average operations per sample for each T functional
C_P	the average operations per sample for each P functional
C_Φ	the average operations per sample for each Φ functional

Table 3.1: Trace transform computational parameters

the functional along the line. Algorithm 1 shows a pseudo-code version of the Trace transform. Other parameters referred to in this section are shown in Table 3.1.

Algorithm 1 Trace Transform Algorithm

```
for func= 1 to N_T do
  for φ = 0 to n_φ − 1 do
    for p = 0 to n_p − 1 do
      for t = 0 to n_t − 1 do
        process pixel using func
      end for
      store result for (φ, p) point
    end for
  end for
  store result for func
end for
```

There are two main steps in computing the Trace transform. The first is to extract the necessary pixels from the source image, given values (ϕ, p) and the second is the actual computation of the Trace results. For each of n_ϕ angles

and n_p lines per angle, let C_ϕ represent the number of operations required to compute the pixel addresses for a line. To compute each of N_T functional results, n_t points on each of the n_p lines for each of the n_ϕ angles must be processed. If C_T denotes the number of operations required on average per pixel per functional, then a total of $n_\phi n_p n_t N_T C_T$ operations are required to compute the traces for an image, while $n_\phi n_p C_\phi$ operations are required for the line extraction.

If triple features are to be extracted, then for each of the N_P diametrical functionals, n_p points must be calculated at a cost of C_P operations per point. This must be applied to each of the N_T trace images produced in the previous step. Finally, N_Φ circus functionals must be computed on n_ϕ points at a cost of C_Φ operations per point. This is applied to each of the $N_T N_P$ vectors from the previous step.

This gives a total number of operations of

$$n_\phi n_p C_\phi + n_\phi n_p n_t N_T C_T + n_\phi n_p N_T N_P C_P + n_\phi N_T N_P N_\Phi C_\Phi \tag{3.10}$$

For an $N \times N$ image, these parameters may take values as follows: $n_\phi = 180$, $n_p \simeq N$ and $n_t \simeq N$. N_T, N_P and N_Φ could take values of 10 each. This would give a total computation complexity of

$$180NC_\phi + 1800N^2 C_T + 18000NC_P + 180000C_\Phi. \tag{3.11}$$

It is thus clear that the complexity of computing the line coordinates and the diametrical functionals are linear with respect to the image size. The complexity of the circus functionals is independent of image size. Meanwhile the first computation step – computing the trace images – is quadratic with respect to image size. For $N = 10$, all three functionals consume equal processing power. For more realistic image sizes of over 100×100 pixels, the trace computation step dominates the processing requirements.

The amount of time taken for C_ϕ and C_T would depend on the implementation; in hardware, it might be possible to do these operations more efficiently than in software. Furthermore, by parallelising in the number of angles (n_ϕ), the number of lines (n_p) and the number of functionals (N_T), the total runtime can be reduced significantly. The key to accelerating any algorithm in hardware is to identify the inherent parallelism then exploit it in the design. By harnessing this parallelism, the algorithm could be accelerated to run in real-time. Given the pseudo-code presented, and these parameters, it is clear that there is significant parallelism in ϕ, p, and N_T

Given these results, it is clear that accelerating the first step, that of trace image generation holds the key to significant speedup. Couple this with the fact that some applications do not require the extraction of triple features and it is clear that this step is most important when it comes to hardware acceleration.

It is also important to note that the Trace transform functionals can in themselves be very complex. While a Radon transform will just use the sum

of pixel values, the Trace transform can apply a variety of mathematical functions. Table 5.2 in Chapter 5 shows the functionals used in a face authentication application, and shows the typical complexity to be expected.

3.7 Related Hardware Implementations

The work presented in this thesis and in [FBCL06] and [FBCL07] is the first to deal with hardware implementation of the Trace transform. The most closely related work is that which deals with the Radon transform and so that is what will be considered here.

In [SCHA92], a system based on four parallel DSP processors for computation of the Radon and Inverse Radon Transform is presented. The parallelism of angles is exploited to increase performance. Different interpolation techniques are compared, and while Linear Interpolation is shown to be slower than Nearest Neighbour, it is chosen due to the increase in quality.

In [BY92], the authors use progressively larger line segments to approximate the line sums, thus significantly reducing processing time. The authors of [FVS05] further develop this algorithm, presenting a hardware implementation that can process 21 512×512 pixel frames per second.

In [SI94], the presented implementation first maps an image to the Polar coordinate system, then uses this to transform it to the Radon domain. The system only deals with binary images. The authors also suggest parallelisation in the angles.

In [MB04], the authors make use of the Radon transform's relationship with the Fourier transform. Using efficient implementations of the FFT and IFFT, they are able to accelerate computation of the Radon and inverse Radon transforms.

Finally, in [CA05], the authors present two architectures for the acceleration of the Finite Radon Transform. They mention the clear distinction between the Finite Radon Transform and the Discrete Radon Transform. The theory is thus distinct.

There are a few important notes to be mentioned, that preclude much of this previous work from being useful in regard to the Trace transform. Firstly, Radon transform implementations assume the function to be applied to each line is a sum. For the Trace transform, this is clearly not the case, so ideas such as partial results and the summing of line segments (as in [FVS05]) cannot be applied. The work in [SI94] actually links closely to the idea of using rotations instead of line extractions that will be shown in the next section. Furthermore, the Trace transform does not retain the mathematical properties of the Radon transform nor its relationship to other transforms, hence work such as that in [MB04] cannot be adapted for a Trace transform architecture.

3.8 Summary

Having discussed the Trace transform in depth, it is clear that it is a very powerful tool in the domain of computer vision. It has shown excellent per-

formance in a wide range of different application domains and its flexibility suggests that it could be used in many more. The computational complexity of the algorithm is an issue though, and applying the algorithm in real time presents a challenge. The algorithm can be thought of as a fusion of two stages. Firstly, there must be some mechanism for extracting lines of pixels from an image. The next part is more complex, and perhaps the core of any implementation; the functionals. Key to the algorithm's wide array of uses is the flexibility in selecting functionals. Furthermore, the functionals are mathematically varied and often complex. With these two attributes in mind, it is clear that hardware accelerations using FPGAs is ideally suited to implementing the algorithm. Hardware acceleration would involve in-depth acceleration of the functionals as well as introducing some degree of flexibility, something which FPGAs are very suited to. A hardware implementation of the Trace transform would be applicable to the Radon and Hough transforms as well, and so be useful for any transform in this class.

Significant acceleration would allow not just fast runtime in a specific application, but also a more thorough method for finding applicable functionals. By offering significant acceleration and a wide array of functionals through a generalised, flexible framework, the designer is free to search a large functional space to find those functionals that offer the best performance for any given application.

Chapter 4 deals with the development of an extensible hardware implementation of the Trace transform that achieves real-time performance. Chapter

5 deals with introducing flexible functionals that can be used to explore the algorithm's performance in a wide variety of application domains.

Chapter 4

A Hardware Architecture for Trace Transform Implementations

4.1 Introduction

In Chapter 3, the Trace transform was introduced along with a discussion of some of its applications, and the computational issues associated with implementing the algorithm. This chapter investigates how the Trace transform can be accelerated in hardware and propose an extensible architecture that offers significant speedup over software implementations. By investigating the algorithm and its computational requirements, a hardware implementation can be created, that provides for significant acceleration. Field Programmable Gate Arrays (FPGAs) bring new opportunities to digital circuit designers, simpli-

fying the hardware design process while allowing for rapid prototyping and testing.

The Trace transform has thus far been applied primarily to images. Furthermore, investigation of functionals has been somewhat limited. By accelerating the algorithm in hardware, and achieving real-time performance, the algorithm could be applied to video sequences too. Acceleration would also open the door to examining a large set of different functionals as applied to different target applications.

In this chapter, a hardware implementation of the Trace transform is presented. A discussion as to the different aspects of hardware acceleration is presented, coupled strongly with a consideration of the appropriate use of heterogeneous FPGA resources. A complete implementation with three functionals is evaluated in terms of speed and area requirements. The implementation presented here achieves a significant speedup of 75× over an equivalent software implementation for three functionals. Part of the work detailed in this chapter has been published in [FBCL06].

4.2 From Algorithm to Architecture

As mentioned in Chapter 2, accelerating an algorithm in hardware is usually approached from two facets. The first is to efficiently implement the complex computations in the architecture such that they can be computed at high speed, forgoing the need for splitting up such instructions as is typical for

software. The second is to exploit parallelism: where a set of instructions must be computed numerous times, while iterating over a parameter, a hardware implementation can implement multiple iterations in parallel. This chapter will deal with exploiting the algorithmic parallelism inherent in the Trace (as well as Radon and Hough) transform, and the design of a real-time hardware system for this class of transforms.

4.2.1 Partitioning into Blocks

As alluded to in the previous chapter, the Trace transform consists of two fundamental building blocks. The first of these takes values of ϕ and p as an input, and returns the pixel intensities along the corresponding line. The second part takes these pixel intensities and applies some functional to them to produce the Trace image. The first part is common between the Trace, Radon and Hough transforms, while the second part is more general and incorporates the facility to compute all three[1].

In a hardware architecture, other blocks deal with dataflow and control; these are detailed in the next section.

[1]The only difference between the Hough and Trace transforms is that the source image in the case of the Hough transform is an edge-map. Since, the result of applying the Radon transform to an edge-map is equivalent to the Hough transform, the names are used synonymously in this thesis.

4.2.2 Exploiting Algorithmic Parallelism

The discussion on computational complexity in Section 3.6 gives an idea as to the areas of parallelism in the design. It concludes that the most significant speedup is to be gained by accelerating the Trace image generation as opposed to the subsequent steps of triple feature extraction. As shown in the aforementioned section, the total computation complexity of the algorithm is given by (3.10), reproduced here:

$$n_\phi n_p C_\phi + n_\phi n_p n_t N_T C_T + n_\phi n_p N_T N_P C_P + n_\phi N_T N_P N_\Phi C_\Phi$$

The last two terms are less significant for images of standard size, and not used in all implementations, and so can be ignored. It is clear that there are multiple candidates for parallelisation. Firstly, a system can be developed that extracts lines for multiple values of ϕ simultaneously. Similarly for values of p. Finally, computing multiple of the N_T functionals in parallel would also yield a significant speedup. For the moment, C_T, the per-pixel cost of functional computation, is ignored. The intention is to design all the functionals with as high a throughput as possible. This is achieved by fully pipelining the functional blocks. This means that they produce a result in each cycle, and thus run as fast as the full system allows. By designing in such a manner, functionals with different computational requirements do not adversely affect the performance of the overall system.

4.3 The Target Hardware

The target platform for this implementation is the Celoxica RC300 Development board [Cel]. This is a widely available board that hosts a Xilinx Virtex II XC2V6000 FPGA, alongside a vast array of peripherals. The only other components of interest for this implementation are the on-board pipelined ZBT SRAMs - there are four 8MB chips - and the USB connection to a host computer. The RAMs can be accessed in pipelined mode, accepting a single read or write instruction per cycle and are 36 bits wide. The Virtex II FPGA on the board has a large logic capacity as well as providing 144 hard-wired multipliers and 144 18Kb BlockRAMs. The system was designed and implemented using Celoxica Handel-C, a high-level C-syntax based hardware description language [Cel]. As mentioned in Chapter 2, high-level languages such as Handel-C allow the designer to write mostly standard C and compile it to an FPGA design. However, the key to exploiting the full power of the FPGA in a design is to write the Handel-C code in a manner that suits the hardware implementation. As such it is important to design the architecture conceptually, rather than attempt to make small modifications to a software version of the algorithm.

4.4 Hardware Architecture

4.4.1 System Framework

Before discussing the details of the hardware implementation, it is necessary to look at the overall system in its constituent parts, as shown in Figure 4.1. A host PC captures image data by way of a standard USB camera. The image data is pre-processed, including resizing and grayscale conversion, before being sent to the development board via a USB interface. The image data is stored in one of the on-board SRAMs before any processing occurs. As each frame becomes available in the SRAMs, the FPGA reads the frame and computes the results. These results are stored in another SRAM, from which the result is transmitted back to the PC, again via the USB interface. On the PC, the data is reorganised and used in the subsequent processing steps of any overall vision system. Hence the crux of the hardware implementation deals with the data between the input and result RAMs. (The FPGA is actually used to control the communication with the PC as well as the data reading and storage.)

A Top-Level Control block oversees the communication between separate blocks and ensures synchronisation. The Rotation Block reads an input image from the on-board RAM and produces a rotated copy at its output. Each Functional Block takes the pixels in this rotated image and uses them to compute the relevant results for each line crossing the image at that angle. This is the first example of parallelism in the design; these functional blocks work in parallel thus producing all their results in the time it would normally take

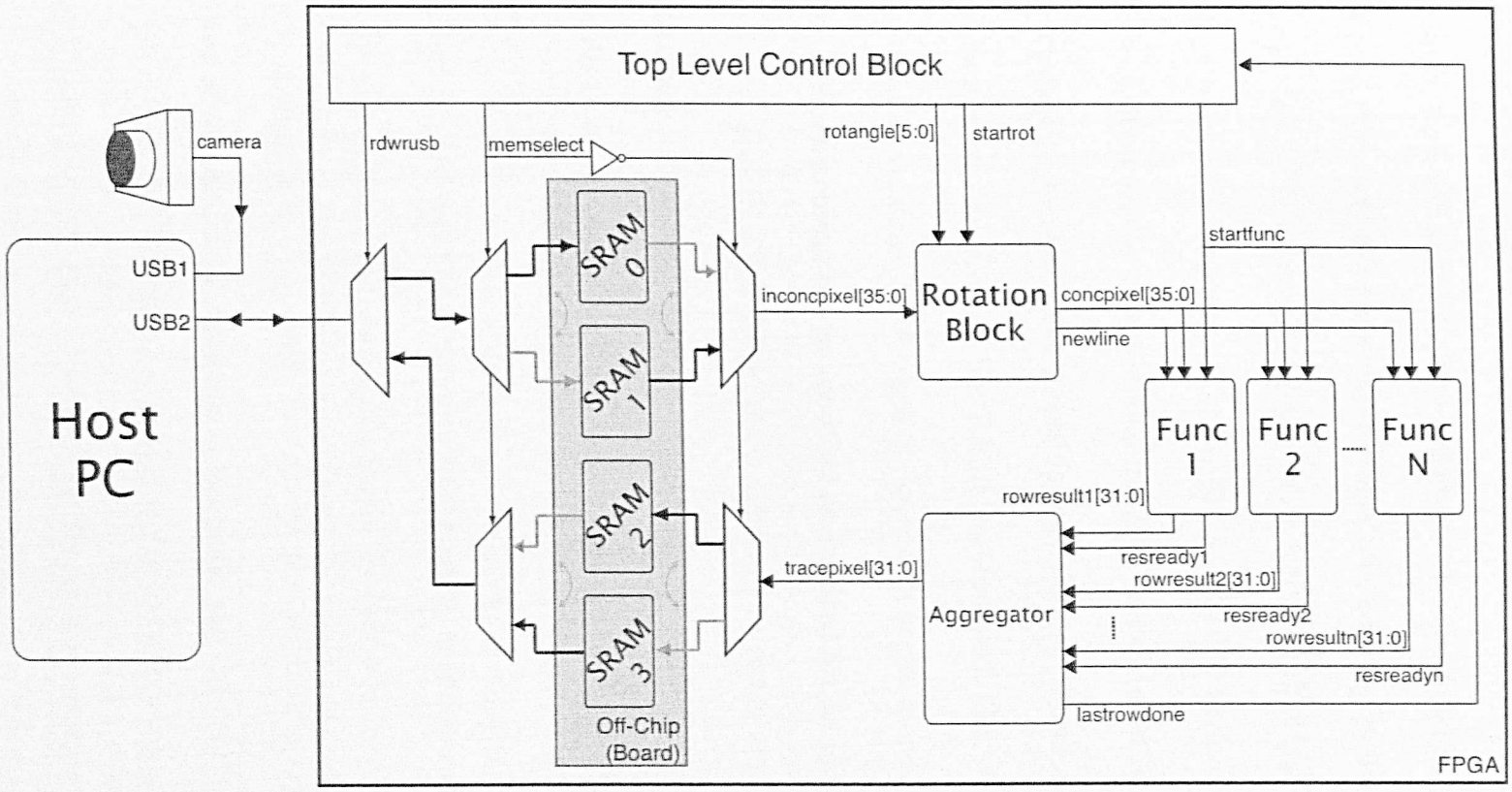


Figure 4.1: Trace Transform Hardware Architecture Overview.

to produce a single functional result. Further parallelism will be examined in subsequent sections. Finally, an Aggregator Block reads the results from each functional in turn and outputs them serially to the result RAM.

The host PC sends the image data, frame by frame to the image RAM on the board. The FPGA reads this image data from the USB port and selects a RAM to write the image to. The RAMs are double buffered to increase performance. This means that while an image is being loaded into one RAM, the other RAM in the pair is being used for calculation. When one calculation cycle is completed, the roles of the RAMs are swapped. The rotation block produces rotated versions of the original image with angles increasing by 2° per iteration. This increment can be modified as required for an implementation. The reason for selecting this value, was to attempt to retain a similar amount of information in the trace domain representation as in the image domain. This results in a mapping of a $256t \times 256$ pixel image to 256×180 points in the transform domain.

The functional blocks read the resultant stream, keeping track of the row beginnings and endings, and passing the results for each row to the Aggregator. Each row corresponds to a single line across the original image. Once the calculation of all lines for all rotations is complete, and the results stored in an output RAM, the host PC reads from this RAM, while the system writes the next set of results to the other output RAM. The results can then be extracted and organised on the host PC for further processing.

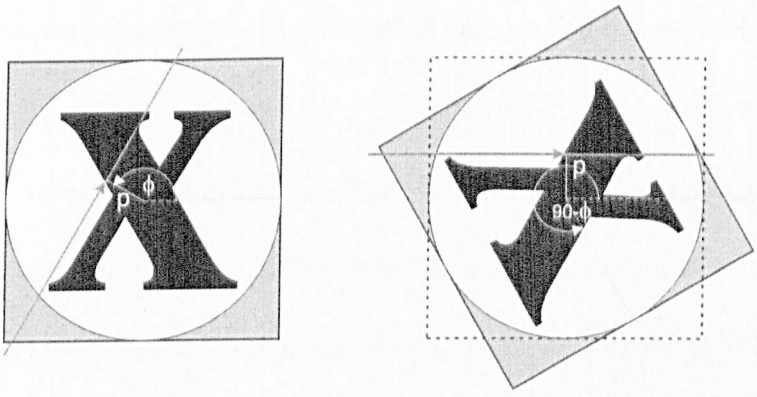
4.4.2 Top Level Control

The Top-Level Control block oversees all the other blocks. It initiates the rotations and ensures that each new rotation is synchronised with the completion of results calculation for the previous rotation. It also manages the double-buffering of the external RAMs.

4.4.3 Rotation Block

Conceptually, the first step in the algorithm is a line tracer: a block that takes a (ϕ, p) value and produces the coordinates of the relevant pixels in the source image. These coordinates are then used to extract pixel intensity values from the source image, stored in off-chip RAM. It is worth noting however, that to compute a trace, all values of (ϕ, p) must be used, hence, the whole image is traced for all angles and at all distances from the centre. Bearing this in mind, a simplification can be made that makes for a more efficient implementation. Rather than iterate over values of ϕ and p, it is possible to rotate the whole image by an angle θ and sample pixel values along the horizontal rows in that rotated version. This would be equivalent to computing all the trace results (all values of p) for a fixed value of $\phi = 90 - \theta$. This equivalence is illustrated in Figure 4.2.

It is necessary here to mention an important caveat. When rotating an image, some parts of the image fall outside the original boundary of the source image. The rotated version must either be larger than the source image, to



(a) (b)

Figure 4.2: Rotating an image then reading across its rows can replace the address generation required for extracting line pixels. The line shown in (a) is equivalent to the row of the rotated image in (b). The shaded area shows the part of the image that will fall out of the image frame for some rotations, and so must not contain an area of interest.

accommodate this extra information, or else the information is lost. An equivalent amount of empty canvas is also introduced from areas "underneath" the image that become exposed. If a square image of size $N \times N$ is rotated through all angles from zero through 360°, then only the portions of the image that fall within a concentric circle with diameter N would be present in all possible rotations. This is also shown in Figure 4.2. It is worth recalling, as mentioned in Chapter 3, that the Trace transform has only been shown to perform well with masked images in the presence of noise [KP01]; since the lines that trace the image may include part of the background too, a lack of masking would allow the background to contribute significantly to the functional results. Due to this fact, it is necessary for the object of interest to be masked; that is, that a binary overlay be present that determines whether or not the corresponding pixel in the image is used in calculations. It is thus a fair assumption for this

system to require any object to fall within the aforementioned area and to be masked appropriately.

To understand why the change from line extraction to rotation simplifies the system, consider first the initial approach. A block would be required that takes a (ϕ, p) value and outputs a vector of addresses. To do this, each line must have a starting point (which must be computed), that may well fall outside the image coordinates. A counter must then be incremented for both image axes and coordinates that fall outside the image must be discarded. The lines will vary in length, and so, some way of tracking the position of the perpendicular to the centre is needed. Furthermore, some way of tracking the correct (ϕ, p) values for each line is required, since each line is extracted independently. Due to the variation in line lengths, there is the added problem of reading from the image RAM inefficiently. Alignment of vectors to take account for the gaps in between readings would also be necessary. A detailed description of the various parameters required to implement the line extraction in this way is given in [MXS07].

Now consider the approach where the whole image is rotated. Each rotation produces a set of all p values for the given rotation (note the offset mentioned above). Since the resultant rotated image is produced in raster scan format, there are no timing gaps, and the vectors are all aligned. This means that no further logic is required before reading from the source RAM. Dealing with a fixed line length of N also simplifies tracking of the p values in the functional blocks, since this is simply the row number in the rotated image (again there

is an offset). Once a rotation has been completed, the source image is again rotated by a new angle, and this produces the vectors for another value of ϕ and so on. This technique simplifies the system significantly and allows for full pipelining of the architecture.

The Rotation Block thus takes an angle as its input and produces the raster scan of the source image rotated by that angle at its output. The source image is read out of order, and since the output is in order, there is no need for image buffers or further logic, since all addressing is inherent in the data.

So far, this modification has dealt with data handling. To fully harness the power of hardware implementation, it is also worthwhile to look at exploiting parallelism in the algorithm. Since results for one angle, ϕ, are in no way computationally related to other values of ϕ, the algorithm could be said to be independent in ϕ. Hence a number of parallel angles could be computed, equal to the number of angles considered. It is however necessary to consider datapath limitations. Three possible methods of parallelising rotations are depicted in Figure 4.3.

For each rotation that occurs in parallel, separate accesses would be required to the source image, due to the out of order access imposed by the design specified above. Hence, multiple copies of the image in separate RAMs would be required, each addressed by separate rotation engines. Since the board RAMs on the target development board only provide single port access, this would mean that each rotation engine would need its own board RAM. Given that there are only four RAMs on the board and two of them are re-

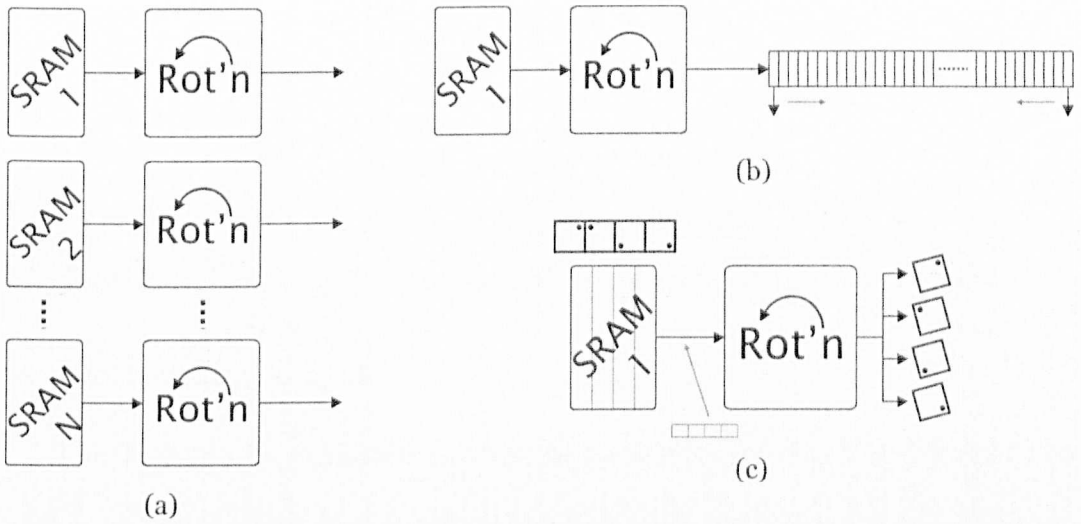


Figure 4.3: Three methods for parallelising rotations. In (a), separate rotation engines run in parallel. In (b), each line is stored in a line buffer and read in both directions. In (c), four orthogonal rotations are concatenated, and thus each iteration of the rotation engine gives four different rotations of the source image.

served for the results, this is not feasible. Another way to parallelise rotations is to read each line into a line buffer, then read that buffer from both ends independently. This would give the lines for both (ϕ, p) and $(\phi + 180°, -p)$ simultaneously.

There is however, another way of enhancing performance even further. Consider that any rotation by a multiple of 90° is simply a rearrangement of data (or alternatively a reassigning of axes values); this is easily implemented in software. It is also clear that a rotation by any (positive) angle, θ, is equivalent to a rotation by some multiple of 90° plus the remaining angle. Formally:

$$\theta = \lfloor \theta / 90° \rfloor \times 90° + \theta \; mod \; 90°. \tag{4.1}$$

As an example, rotation by 212° is equivalent to rotation by 180° followed

Rotation	x	y
$0°$	x	y
$90°$	y	$N - x$
$180°$	$N - x$	$N - y$
$270°$	$N - y$	x

Table 4.1: Base orthogonal rotation coordinates, for an $N \times N$ image.

by a further rotation by $32°$.

This fact can be exploited in order to parallelise rotations as follows. The source image is a standard 8-bit greyscale image. The external board RAMs are 36-bits wide, and hence, storing a single image is a waste of the available wordlength. Instead, what can be done is to store the four orthogonal base rotations ($0°$, $90°$, $180°$ and $270°$) concatenated in a single RAM word. Since the host computer can easily construct the other three rotations from a standard image, there is no real computational cost to be considered. Table 4.1 shows that any orthogonal rotation can be obtained with no more than simple calculations that can be performed extremely fast on a host PC.

Now when a rotation by angle θ is carried out, the RAM word that is output can be spliced to give the relevant pixel for the rotations by θ, $\theta + 90°$, $\theta + 180°$ and $\theta + 270°$. This effectively quadruples performance with only minimal impact on rotation block area. The area impact is only as a result of increasing the size of the registers.

A new rotation is complete every $65,536$ cycles plus a few cycles used to fill and flush the pipeline. With a rotation angle step size of $2°$, 180 rotations are needed for a full set of results. Since 4 rotations are computed in parallel,

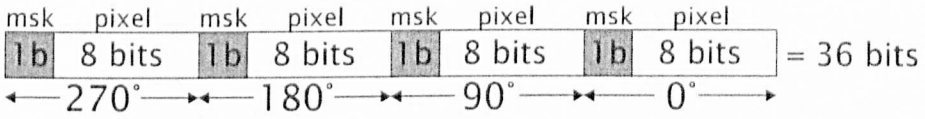


Figure 4.4: Structure of a single word in external RAM.

the actual number of rotation operations is 45. Hence, assuming a rotation latency of 65,560 cycles (to include the margins mentioned above) a full set of rotations is complete in just under 3 million clock cycles.

The construction of the orthogonal rotations occurs on the host PC, and only once per frame. While this takes $4 \times N^2$ reads on the PC (ignoring the effects of caching), the resultant concatenated image is rotated 45 times instead of the the 180 that would be needed for a standard image, in order to construct a full trace. This saves over 9 million cycles per trace, at a cost of $3 \times N^2$ extra cycles on the PC (for which cycle times are much shorter). Hence the overhead is minimal.

Since all images are also masked, the four mask rotations are also stored in the RAM. With four 8-bit image words and 4 1-bit masks, the total wordlength is 36-bits which matches the RAM perfectly. The makeup of a single RAM word is shown in Figure 4.4. Loading an image onto the board over USB requires the data to be sent in single bytes, this means a 256×256 pixel image takes $256 \times 256 \times 5 = 327,860$ cycles to be transferred from the PC to the board RAM.

In order to compute the rotations, the system proceeds as follows. The input angle is used to address sine and cosine lookup tables (stored in Block-RAMs). The resultant values are then used to compute the standard Cartesian

rotation equations:

$$x' = x\cos\theta - y\sin\theta \tag{4.2}$$

$$y' = x\sin\theta + y\cos\theta \tag{4.3}$$

The calculations are carried out using 8-bit fixed-point calculation. Nearest neighbour approximation is used whereby each sample point takes the value of its nearest pixel; this avoids the more complex circuitry and scheduling required for bilinear or bicubic approximation. The result is that a complete rotation of an $N \times N$ image is complete in N^2 clock cycles.

The x and y coordinates obtained from this computation are used to read specific pixels from the source image in off-chip RAM. Recall that these source pixels are in fact a set of four concatenated orthogonal base rotations. When one of these concatenated "pixels" is read, it is spliced into its four separate parts and each of those is used to build a separate rotated image. This means that in N^2 cycles, 4 separate rotations have been completed.

The resultant rotated images stream through the system in raster scan format. This is where each row is transmitted, one pixel at a time, followed immediately by the next row and so on. This makes the subsequent blocks simpler since there is no complex buffering or ordering to be considered. By parallelising rotations in this way, the rotation engine now only needs to iterate 44 times (for a 2° step size) for a full set of rotations, as opposed to the 178 iterations originally needed.

The overhead imposed by this method of parallelising is primarily at the host PC. Rather than sending an image directly requiring a single read from memory and a single USB write, the system must now read from memory four times for each pixel, transmit four bytes rather than one, hence four times the amount of image data is transferred over USB. This overhead does not affect the performance of the system, however, as it is done in parallel with the system processing a frame.

4.4.4 Functional Blocks

The actual computation of results for the Trace transform occurs in the functional blocks. Each block must process the output of the rotation block, computing the results of the appropriate calculation applied to each row of input, and then pass this result on to be stored in the result RAM. A few aspects of the implementation should be noted here. Firstly, the input to the functional blocks is the raster scan of the rotated image. To assist in keeping the data aligned, a "new row" signal is also passed by the rotation block. Each new rotation is processed independently, so there is a short gap in processing between subsequent rotations. This is needed to allow for functional blocks with different latencies to process the correct data at the same time.

Secondly, recall that the output of the rotation block is four orthogonal rotations rather than one. Each functional block therefore splices this data and computes results for each of the four rotations independently. This means that all the computation datapaths and registers are duplicated four times.

The control circuitry is kept combined for compactness and synchronisation purposes.

Finally, it is important that each functional block makes use of the mask associated with each pixel to decide whether it is used in the calculation. Recall that masked-out pixels are ignored in the Trace transform. The three functionals presented here do not rely on the position of input pixels for computing the result. However, a counter is maintained that keeps track of the position in the row while it is being processed. This serves to mark the end of each row.

When the results for each row are ready, they are passed in parallel to an output buffer and a "result ready" signal is passed to the Aggregator which deals with storing the results to off-chip memory. Since the board SRAMs have a 36-bit wordlength, the widths of the datapaths are tailored to ensure that the final result fits within this limit. In the case of the three functionals presented here, the range of results fits with no need for any scaling. Where some scaling is required, the impact on accuracy must be considered.

In developing this extensible architecture, the three functionals shown in Table 4.2 were implemented as a proof of concept. They were built fully pipelined, such that they could be changed without impacting the timings of the whole system. In Chapter 5, a framework for developing flexible functionals to replace these ones is presented.

Details of the design for each of the functionals are shown below. The "D" blocks in the diagrams are registers. The "Store" block is activated at the end

No.	Functional
1	$T(f(t)) = \sum_0^N f(t)$
2	$T(f(t)) = \sum_0^N \lvert f'(t) \rvert$
3	$T(f(t)) = \left[\sum_0^N \sqrt{\lvert f(t) \rvert}\right]^2$

Table 4.2: Trace functionals used in this implementation.

of each line, storing the final result for that functional.

Functional 1

This is the simplest of the functionals, summing all the pixels in each trace line. The Trace Transform using this functional is the equivalent of the Radon Transform. The corresponding equation is shown in (4.4). Figure 4.5 shows the schematic diagram of the design.

$$T(f(t)) = \sum_0^N f(t) \tag{4.4}$$

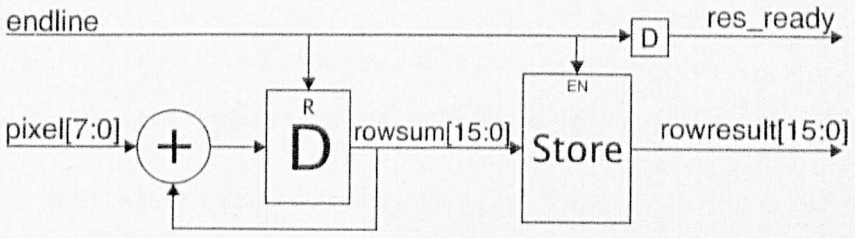


Figure 4.5: Schematic diagram of Functional 1.

Functional 2

This functional sums the absolute differences between adjacent pixels in each trace line. The corresponding equation is shown in (4.5). Figure 4.6 shows the schematic diagram of the design.

$$T(f(t)) = \sum_{0}^{N} |f'(t)| \tag{4.5}$$

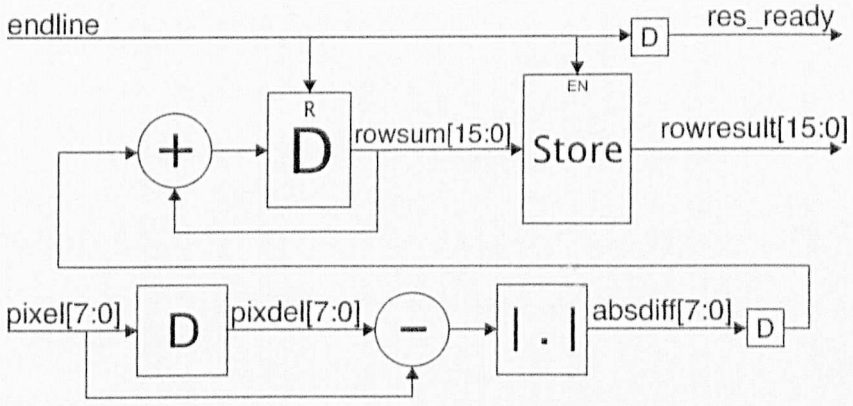


Figure 4.6: Schematic diagram of Functional 2.

Functional 3

This functional is the square of the sum of the square roots of the pixels in each trace line. The square root was implemented using a lookup table since the pixel intensities are only 8 bits wide. The square operation was implemented using the embedded multipliers. The corresponding equation is shown in (4.6).

Figure 4.7 shows the schematic diagram of the design.

$$T(f(t)) = \left[\sum_{0}^{N} \sqrt{|f(t)|}\right]^2 \tag{4.6}$$

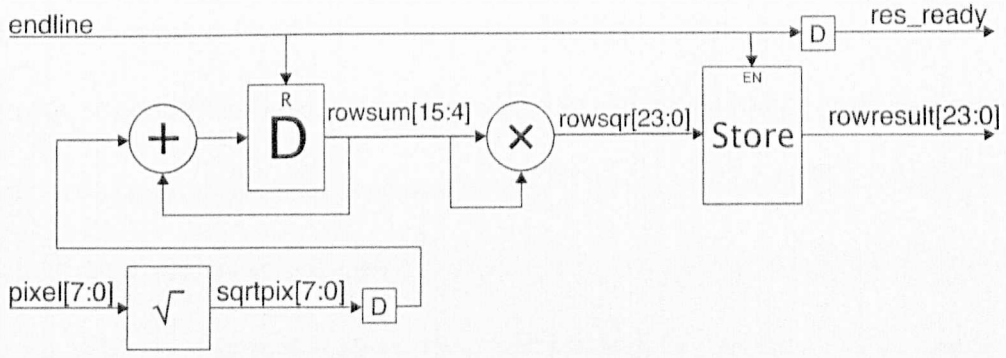


Figure 4.7: Schematic diagram of Functional 3.

4.4.5 Aggregator

The Aggregator polls the functionals in a round-robin fashion awaiting a "new result" signal[2]. When received, the aggregator proceeds to store the four results from the current functional in a serial manner. This is done to avoid having a large data bus between each of the functionals and the aggregator. Since there is only a new result every N cycles (256 in this implementation), there is sufficient time to read each result from each functional in series. The results are stored in an on-board RAM addressed using a concatenation of functional number, rotation and row number. The contents of this RAM can then be read

[2]It is preferable that the polling is ordered such that the functional that finishes first is the first polled. This would provide the most efficient use of the storage time window.

by the host PC over USB and the results used in further stages of processing.

4.5 Platform Considerations

The architecture presented here suits any target hardware platform. While other development boards do offer more significant throughput to external resources, it simplifies the architecture and control considerably, to limit accesses to external communication and memories. The system presented is effectively a block that sits between two memories, processing the contents of one, using no external communication, save the reading of the data, and storing its results in another memory. Stream processing is what facilitates the simple architecture, enabled by the use of horizontal lines from the rotated image, that stream through the system in order.

The architecture could be synthesised on any of the modern FPGAs available today. They all contain embedded memories and multipliers. Indeed with some of the latest devices, the resource usage would decrease, while the speed would be significantly increased. An important limiting factor that must also be considered beside the FPGA is the speed of the external memories. In this implementation, this proved more restrictive than the FPGA itself.

The use of the wide external memories on the development board enables the parallelisation of rotations in the manner presented. If such resources were not available, the alternative method of reading a line into a buffer, then processing from both directions (as shown in Figure 4.3(b)), could be used.

Alternatively, if more external memories, or memories with more ports are available, separate rotations engines could be employed to the same effect. The use of separate memories or multi-ported memories would enable more than four parallel rotations, while the current method cannot be improved upon as it stands.

The architecture is in itself efficient and flexible and could be easily modified to suit a different target platform.

4.6 Implementation Results

This design was implemented on the development board described in Section 4.3 using the Handel-C language and Celoxica DK tools[Cel]. The implementation instantiates three functionals as detailed above, however, the design is modular so that adding extra functionals is straightforward. Synthesis results are shown in Table 4.3. The resultant clock-speed of 80MHz is limited by the development board libraries used to access on-board resources. This high speed was primarily due to the fully-pipelined nature of the design. Significant use of the channel communications provided for in Handel-C was made.

This clock speed results in a throughput of 26 frames per second for a 256×256 pixel image. In comparison, a highly optimised MATLAB equivalent in software, running on a Pentium-4 2.2GHz, took just over 3 seconds to complete the same calculations on a single frame. This hardware architecture thus gives over a 75 times speed-up. This acceleration increases with the number

Synthesis Results	
Device	Xilinx Virtex II XC2V6000-6
Clock Speed	80MHz
Frame Rate	26fps
Slices	2,070 (6%)
BlockRAMs	6 (4%)
Embedded Mults	8 (6%)

Table 4.3: Trace transform architecture synthesis results for the Celoxica RC300 Development Board.

of functionals, as additional functionals would slow down the software version while not affecting the speed of hardware implementation. Bear in mind also, that other functionals can be significantly more complex, and hence the effect of hardware acceleration can be more pronounced in those cases.

Figure 4.8 shows the system-level timing for the architecture. Consider a 256×256 image, so $N = 256$. In this case, the architecture completes four orthogonal rotations of the image in 65,536 cycles. Each of these 65,536 pixels is passed to the functional blocks in order, with a new row starting every 256 cycles. In the last cycle of each row, the functional copies the results for each of the four orthogonal rotations to output buffers and continues with the next row. The Aggregator waits for a result to arrive at the first functional. It takes 7 cycles to store the result for the four rotations into the output RAM, then it continues with the other functionals in order, storing each of the results in the external RAM. It is available until the next results arrive, 256 cycles after the previous ones. Once the last result is stored for a rotation, the Aggregator instructs the rotation block to start another rotation, and so the

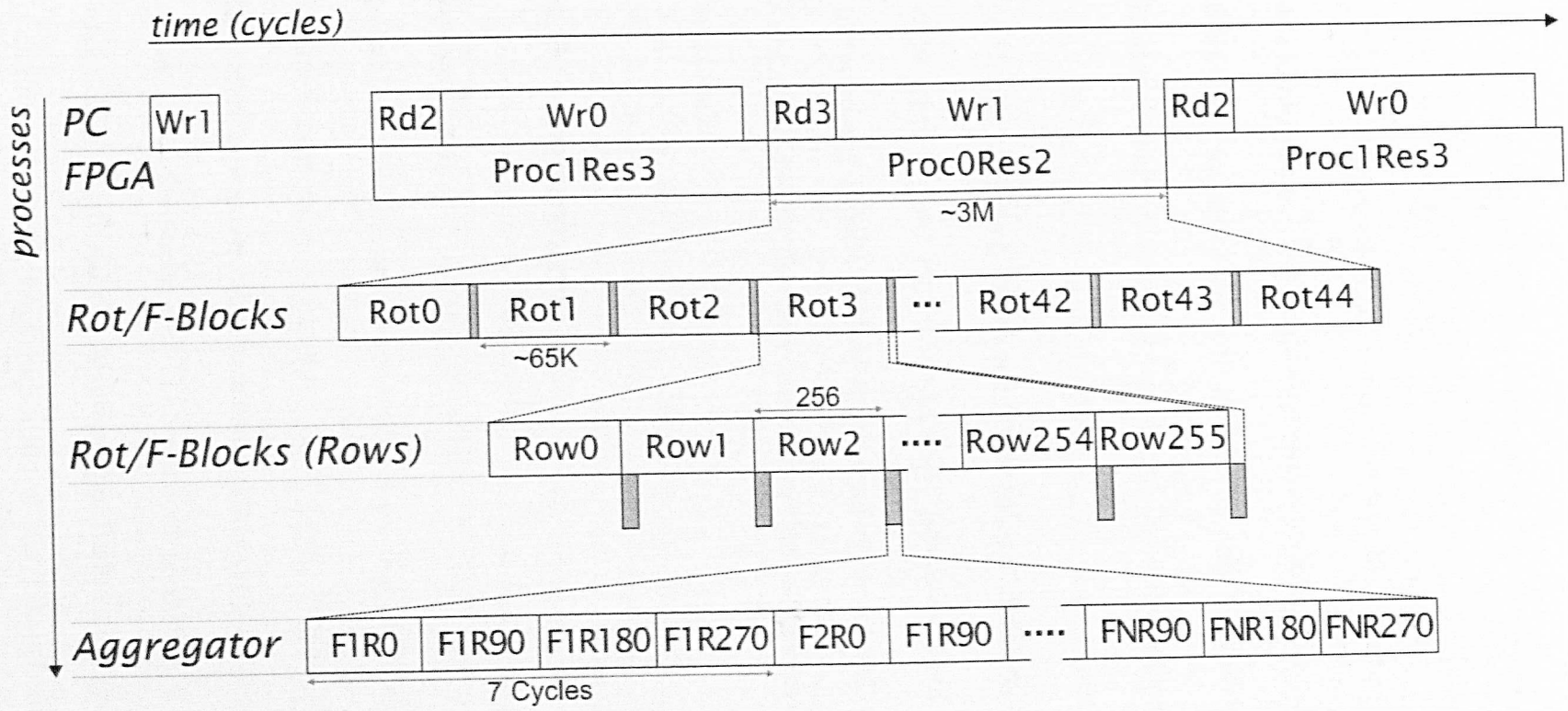


Figure 4.8: Details of the system-level timing of the design. The uppermost portion shows the double-buffering in action. The PC reads results from the previous iteration, then writes image for the next iteration. At each stage, the FPGA is reading from and writing to the opposite RAM in each pair. The Rotation and Functional Blocks process one (4-way) rotation at a time, with small gaps to allow for the pipeline to flush. Rows are processed in straight succession, with results stored in parallel. These results are then accessed by the Aggregator serially. Rd2 means read from SRAM2, Wr0 means write to SRAM0, Rot4 means rotation number 4, FNR180 means the result for the third orthogonal rotation for Functional N.

process continues. Hence, a complete computation of the Trace transform for a 256×256 pixel image takes approximately 3 million clock cycles. For a system running at 80MHz, that results in a throughput of 26 frames per second.

The implementation serves as an extensible architecture for Trace Transform applications. The simplicity of the architecture belies its power. More functionals can be added with ease. The only theoretical limit is where the functional result storage latency exceeds the time between successive row results. Given that for a 256×256 image, a new result comes in every 256 cycles, and that each functional result takes 7 cycles to store, this allows sufficient slack in time for 36 functionals. Assuming these functionals are each optimised and pipelined, they will not affect the overall latency of the system or its clock speed. The only limitation, then, is the area requirements of the functionals.

4.7 Summary

By harnessing the inherent parallelism in the Trace transform, it has been possible to design a hardware architecture that offers a significant speedup of 75 times over software for three functionals. Furthermore, the hardware architecture is scalable, able to accommodate the addition of further functionals without a performance cost. The methods used here are generalisable for the whole class of transforms including the Radon and Hough transforms. This is the first hardware implementation of the Trace Transform, and illustrates

the power of a hardware-centric design methodology. The design of this architecture was developed directly from an understanding of how the algorithm works, rather than simply by parallelising the loops in a software implementation. One of the difficulties in designing hardware is the dataflow management; one must consider, given the constraints of the communication channels, how to transport data into and out of the system. One must also attempt to circumvent the need for intermediate storage. The simplification of line-extraction to a series of rotations allowed for a simple datapath using streamed data with no need for internal storage. Parallelising the rotations in the manner described further quadrupled performance with minimal impact on storage needs. This architecture serves as the foundation for building a flexible Trace transform functional exploration framework, presented in Chapter 5.

Chapter 5

Flexible Functional Blocks for Exploration

5.1 Introduction

In Chapter 4, a hardware architecture that implements the Trace transform was described. It is capable of real-time performance, of 26 frames per second, when processing 256×256 pixel images. The architecture was designed with extensibility in mind, and all the blocks are fully pipelined, such that alternative functionals can be swapped in without any change to the architecture or timing. In this chapter, a framework for developing flexible functionals for the Trace transform architecture in Chapter 4 is presented. Three example functionals are also described, that cover those used in the Shape Trace Transform (STT) for face verification [SPKK05]. These can be instantiated in the

aforementioned framework and provide for run-time re-programability[1].

A significant opportunity afforded by accelerating the Trace transform is that of investigating suitable functionals for a specific application. The Trace transform's strength lies in its general definition, allowing for functionals to be selected based on their strength for a specific application. A flexible hardware architecture greatly simplifies the exploration of the functional space. Rather than iterate through a short list of alternatives one-by-one, a large number of alternative functionals can be investigated in parallel without the need to re-synthesise the design. In order to achieve this, the heterogeneous resources on the FPGA, specifically the on-chip Block RAMs, are used to create generalised functional blocks that can compute a selection of different functional results using the same block. Changing the contents of Block RAMs, and configuration registers in each functional block, determines which specific computation is to be computed. In this way, there is no need to re-synthesise the design with each new set of functionals.

It is worth noting that there has not been, to date, a thorough investigation comparing functionals for a given application. Applications of the Trace transform presented in the literature (as summarised in Chapter 3), have simply used the functionals that were available. While work on determining functionals that perform well for specific geometric invariance, such as affine transfor-

[1]Note that the term re-programability is used here to differentiate from FPGA reconfigurability. Runtime and/or partial reconfiguration of FPGAs is still an active area of research, without a mature tool-flow as yet. All the work in this thesis is implemented using static configurations.

mations, has been conducted, functionals tailored to more specific applications have not been investigated. A hardware system, combined with a framework for developing flexible functionals, enables this process and allows the Trace transform to be investigated for many new applications.

5.2 A Framework for Designing Flexible Functionals

In order to achieve some flexibility in the functionals to be implemented, there are three possible approaches. The first is to create multiple functional designs, then select the appropriate ones during synthesis. While such an approach would work, it would mean that each combination of functionals would require a fresh synthesis run. This can be time-consuming, negating the benefits of hardware acceleration. This method also fails to avail of the more advanced methods that modern FPGAs afford.

The second method is to use runtime reconfiguration. Runtime reconfiguration refers to the modification of the FPGA configuration during runtime. Recall that the circuit implemented in an FPGA is defined by a configuration bitstream that details the states of each of the circuit units and the routing configuration. Applying a new bitstream allows the circuit to be changed while running. Partial reconfiguration allows parts of the circuit to be updated while others continue to run. Such a process could be employed to allow functionals to be swapped in and out in various combinations. The design flow is, how-

ever, more complex and the tools are not yet completely suited to this form of design, though it continues to be an active area of research [Sed06].

Another simpler method, which still provides relative flexibility is to design blocks that can compute multiple functionals, selectable by some configuration information. One of the simplest ways to achieve this is to use Block RAMs to compute basic mathematical functions within the functionals. Using Block-RAMs as lookups in this manner means that the actual operation performed can be changed without modifying the circuit. Furthermore, it is possible to use small registers with stored values to determine the selection of paths in a system with multiple datapaths. These two modifications allow for a single functional block to compute a large variety of functionals, and for the selection to be modified on the fly.

5.2.1 Integration into Trace Transform Architecture

It is necessary to summarise the requirements for a generalised functional that can be used in the Trace transform architecture presented in Chapter 4. Recall that the output of the rotation block is a 36-bit signal, as read from the image RAM. This signal is a concatenation of the four orthogonal rotations and their masks. Every cycle, one of these "pixels" arrives at the functional blocks. At the start of each row in the rotated image, a "new line" signal goes high. Each functional block produces a single result per orthogonal rotation for each full row of input. This result is stored to registers, in parallel, once it is ready, and a "result ready" signal is sent to the Aggregator block. The Aggregator block

is set to poll the functional blocks in a round robin fashion starting with the one that has least latency. It is also important to note that subsequent rows within the same image arrive in straight succession with no gaps.

This information can be used to devise a basic framework for a flexible functional block. Firstly, the incoming signal is spliced into the constituent orthogonal rotations and masks. Each of these orthogonal rotations is processed in a separate pipeline, though with unified control circuitry. The first control element needed is a counter to keep track of the current pixel number within the row. An "end-row" signal is extracted from this counter to signify the arrival of the last pixel in the row. The designer must consider the pipeline delays introduced by circuit elements, and delay this end row signal by the appropriate amount to extract a "last result" signal which goes high at the point at which the last pixel of a row has been processed. This is used to enable a bank of four registers that store the functional results for the four rotations in parallel.

In each computational stage, the mask must be taken into account. Those pixels which are masked out should not be incorporated into the computation. Again, it is necessary to have aligned versions of the mask signals so that each point in the pipeline processes the correct mask values. For some computations, the mask need not be applied if the masked out areas are set to zero. For example, an accumulator does not use mask information since summing a zero or not summing is the same. The only assumption is that the input image is set to zero for all masked-out pixels.

Some functionals may require the incorporation of the pixel position in the calculation. The counter mentioned above can be used for this purpose. For this to work, it is necessary to add a single cycle delay to the input "pixel", to ensure alignment, since the counter is reset with the first pixel to arrive.

Table 5.2, to follow, shows the functionals used in a face authentication application of the Trace transform. It is clear that most functionals involve a summation over all the processed pixels. This may be implemented by instantiating an accumulator at the end of each of the four datapaths. The final result is limited to 32 bits in order to fit in the board SRAMs. Hence, the width of the input to the accumulator must be restricted to 24 bits such that the maximum of 256 such inputs can be accommodated. The accumulator is reset by an appropriately aligned version of the aforementioned "new-line" signal. An accumulator may not be needed for other functionals that don't involve a summation.

In order to create the flexibility sought, on-chip BlockRAMs are employed as lookups. A lookup RAM contains pre-computed values for some function stored at each location. For example, for a lookup RAM to be used for computing $\cos(x)$, the contents of each memory location would have to contain the value of $\cos(addr)$, where $addr$ is the address of the RAM word. Thus, when value a is applied to the address input of the lookup, $\cos(a)$ emerges at the data output. Note that the RAMs are in fact used as ROMs in this situation. However they are still referred to as RAMs because they retain their write capability. This write capability is what allows them to be exploited for adding

flexibility. Since these RAMs can be configured with data at runtime, there is no need to re-synthesise the design in order to change the RAM contents, and thus the function computed by each lookup table

Now consider that a wide range of functions can be implemented in this fashion, and it becomes clear that a functional block with lookup RAMs incorporated can implement a wide array of different funcitonals. Some of the arithmetic functions that can be computed in this manner include x, x^2, $\sqrt{x}$, $\ln(x)$, $\sin(x)$, to name but a few. The only limitation is that the input value must be bound since the size of the RAM must be predetermined. A small increase in the range of an input can impact the resources used dramatically. The on-chip BlockRAMs on a Xilinx Virtex II are 18Kb in size, and can be configured in a number of ways, between 512 $\times$ 36-bits to 16K $\times$ 1-bit [Xil99b]. The specific configuration must be set in advance. Fortunately, with most image processing systems the input pixels are constrained to be 8-bits wide. The configuration that corresponds closest to this address width is 512$\times$36bits. However, exploiting this wordlength efficiently would be difficult, given the presence of four datapaths within each functional. The presence of multiple operations within each functional would also mean extremely wide signals at the end of the processing pipeline. Hence the size of each lookup is set to 256$\times$16 bits. While this does not fill a BlockRAM, it uses only one, and provides for 2^{16} levels of precision.

Since each functional has four datapaths, it is expected that four Block RAMs would be needed for each lookup in the functional. In fact, due to

the dual-port configuration of the BlockRAMs, and the fact that the lookup information is identical for each of the datapaths, it is possible for a single BlockRAM to be shared by the datapaths for two orthogonal rotations simultaneously.

It is important to consider the case when there may be subsequent lookup stages within a functional block. This presents a problem since the output of the first lookup stage is 16-bits wide while the input into any subsequent lookup can only be 8-bits wide. The designer must incorporate some way of selecting how this is done. A configuration register controlling a multiplexer can be used to select between different splicings of the datapath as required.

The other facet of flexibility to be introduced is that of variable datapaths. In order to allow for selection, each functional includes an 8-bit configuration register with each bit being used to make a binary selection. This can be to select the upper or lower 8 bits from a lookup output (as alluded to above), or to enable or disable parts of the functional computation.

All of these considerations result in a general framework as shown in Figure 5.1. Combined, these allow a single functional block to compute a number of different functionals without any need for reconfiguration.

5.2.2 Lookup Accuracy Considerations

The use of lookup memories with constrained wordlengths means that accuracy is sacrificed to some extent for the functions approximated. For some functions, the specified wordlengths are sufficient for an exact mapping. Table 5.1 gives

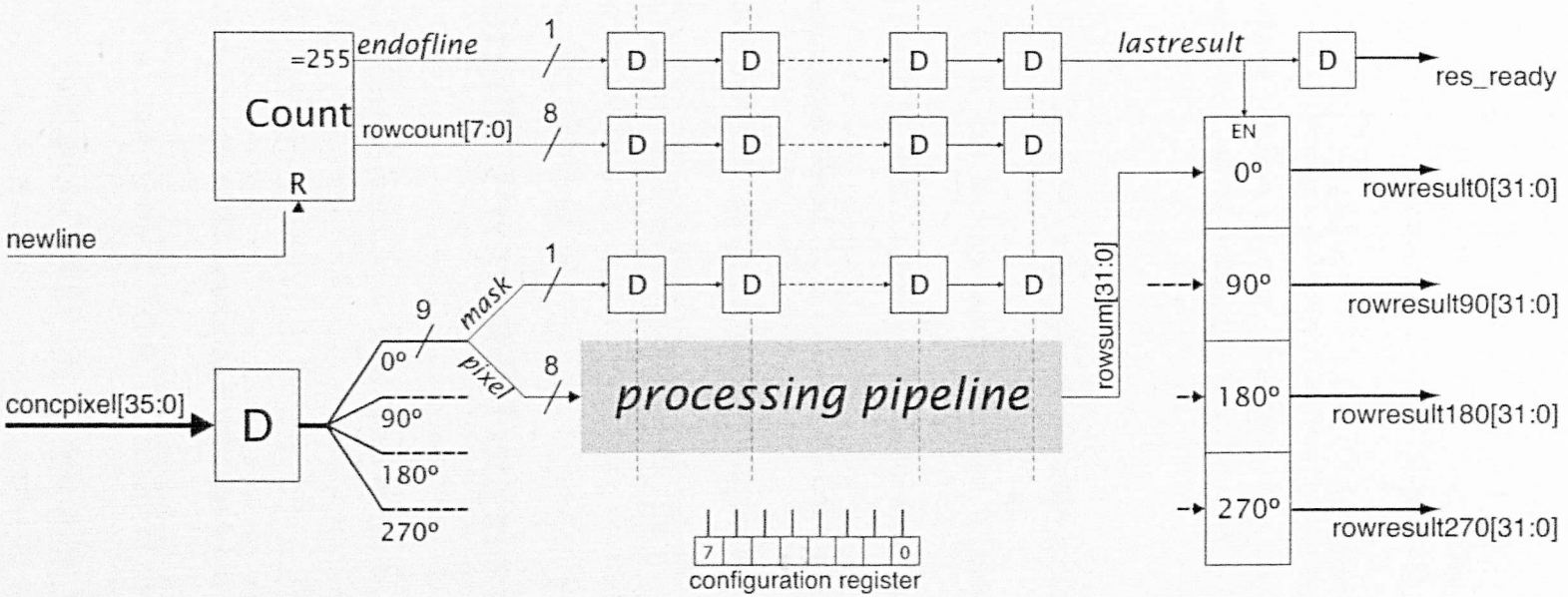


Figure 5.1: The framework for implementing flexible Trace transform functionals. The processing pipeline would contain a collection of lookups, multipliers, adders and other primitives and most likely an accumulator. There would be one processing pipeline for each of the orthogonal rotations. The configuration register would be used for selection between multiple datapaths.

Lookup Function	Mean Relative Error
x	0%
x^2	0%
x^3	2.85%
$\sqrt{x}$	0.00139%
$\ln x$	0.00142%
$1/x$	0.166%
$\sin(x)$	0.011%

Table 5.1: Mean relative error of functions when approximated by a 256×16 bit lookup memory (using maximum possible lossless scaling). x ranges from 0 to 255.

the mean relative error for a number of example functions when implemented using the 256×16 bit lookup memories used in this architecture. If more precision is required, it is relatively straightforward to extend the wordlength. Using a single Block RAM, it is possible to hold 256×36 bits. However, one must consider that this will significantly widen the rest of the datapath, as well as resulting in much wider functional results.

5.3 Initially Proposed Functionals

A number of different functionals have been suggested for the different Trace transform applications discussed in Section 3.5. A detailed list was given in [SPKK05], and consists of 22 functionals used for a face verification application. These functionals are shown in Table 5.2. Many of these functionals were also used in other applications [PK04]. It is clear from Table 5.2 that there are groups of functionals that are very similar in structure, such as numbers 9, 11, 12, 13 and 14, as well as 20, 21 and 22.

No.	Functional	Details
1	$T(f(t)) = \int_0^\infty f(t)dt$	Radon Transform
2	$T(f(t)) = \left[\int_0^\infty \lvert f(t)\rvert^{\frac{1}{2}}dt\right]^2$	
3	$T(f(t)) = \left[\int_0^\infty \lvert f(t)\rvert^4 dt\right]^{\frac{1}{4}}$	
4	$T(f(t)) = \int_0^\infty \lvert f(t)'\rvert dt$	$f(t)' = (t_2 - t_1), (t_3 - t_2), ..., (t_n - t_{n-1})$
5	$T(f(t)) = median_t\{f(t), \lvert f(t)\rvert\}$	Weighted median
6	$T(f(t)) = median_t\{f(t), \lvert f(t)'\rvert\}$	
7	$T(f(t)) = \left[\int_0^{n/2} \lvert\mathcal{F}\{f(t)\}(t)\rvert^4 dt\right]^{\frac{1}{4}}$	$\mathcal{F}$ means taking the Discrete Fourier Transform
8	$T(f(t)) = \int_0^\infty \left\lvert \frac{d}{dt}\mathcal{M}\{f(t)\}\right\rvert dt$	$\mathcal{M}$ means taking the Median over a length 3 window, and $\frac{d}{dt}$ means taking the difference of successive samples
9	$T(f(t)) = \int_0^\infty rf(t)dt$	$r = \lvert l - c\rvert$; $l = 1, 2, \ldots, n$; $c =$median$_l\{l, f(t)\}$
10	$T(f(t)) = median_t\left\{\sqrt{r}f(t), \lvert f(t)'\rvert^{\frac{1}{2}}\right\}$	
11	$T(f(t)) = \int_0^\infty r^2 f(t)dt$	
12	$T(f(t)) = \int_{c^*}^\infty \sqrt{r}f(t)dt$	$c*$ signifies the nearest integer to c
13	$T(f(t)) = \int_{c^*}^\infty rf(t)dt$	
14	$T(f(t)) = \int_{c^*}^\infty r^2 f(t)dt$	
15	$T(f(t)) = median_{t^*}\{f(t^*), \lvert f(t^*)^{\frac{1}{2}}\rvert\}$	$f(t^*) = \{f(t_{c^*}), f(t_{c^*+1}), \ldots, f(t_n)\}$
16	$T(f(t)) = median_{t^*}\{rf(t^*), \lvert f(t^*)^{\frac{1}{2}}\rvert\}$	$l = c^*, c^* + 1, ..., n$; $c = median_l\{l, \lvert f(t)\rvert^{\frac{1}{2}}\}$
17	$T(f(t)) = \left\lvert\int_{c^*}^\infty e^{i4\log(r)}\sqrt{r}f(t)dt\right\rvert$	
18	$T(f(t)) = \left\lvert\int_{c^*}^\infty e^{i3\log(r)}f(t)dt\right\rvert$	
19	$T(f(t)) = \left\lvert\int_{c^*}^\infty e^{i5\log(r)}rf(t)dt\right\rvert$	
20	$T(f(t)) = \int_c^\infty \sqrt{r}f(t)dt$	$r = \lvert l - c\rvert$; $l = 1, 2, \ldots, n$;
21	$T(f(t)) = \int_c^\infty rf(t)dt$	$c = \frac{1}{S}\int_0^\infty l\lvert f(t)\rvert dt$; $S = \int_0^\infty \lvert f(t)\rvert dt$
22	$T(f(t)) = \int_c^\infty r^2 f(t)dt$	

Table 5.2: The Trace Functionals T used in [SPKK05], grouped by similarly structured sets.

For the face verification application, the Trace transform is used in two ways, as mentioned in Chapter 3. Firstly, the traces constructed are directly compared to each other using the Weighted Trace Transform (WTT), but the performance of this method is shown to be mediocre [SPKK05]. The second method is the Shape Trace Transform (STT), where the traces are thresholded and the resultant shapes compared rather than the traces themselves. This method proved much more accurate. Of the 22 functionals listed in Table 5.2, numbers 1, 2, 7, 9, 11, 12, 13, 14, 20, 21 and 22 were found to be useful for the STT.

5.4 Flexible Functional Blocks

In the following sections, three different generalised functional blocks are presented. Each of them can implement a number of functionals listed in Table 5.2 as well as some others obtained by changing the lookup functions within each block. Block diagrams are shown for each type, though for simplification, only a single datapath is shown whereas, as mentioned previously, each functional actually processes four rotations in parallel. The signal wordlengths are shown on the connections. Each lookup is implemented on a single BlockRAM.

5.4.1 Type A Functional Block

This functional block is able to compute functionals 1,2 and 4 from Table 5.2. These three functionals were combined despite being different, due to the

Bit	Effect
0	Subtract delayed sample
1	Select upper 8 bits
2	Apply l_3
3	Square row sum
4-7	Unused

Table 5.3: Configuration register for Type A functional block.

simplicity of the operations involved. A block diagram is shown in Figure 5.2. Note that each circuit element takes a single cycle to run and that the dashed parts are optional, determined by the configuration register. "D" is a single cycle delay register.

The block takes an input pixel and applies a lookup function, l_1 to it. Optionally, function l_2 is applied to a one cycle delayed version of the input pixel and the absolute difference is taken; the actual datapath is decided by the configuration register. Function l_3 can then optionally be applied to either the upper or lower 8 bits of the output from that stage. The result is passed through to the accumulator and at the end of the row, the final result is optionally squared. The configuration register is configured as in Table 5.3. To implement the functionals using this block, the configuration must be set as shown in Table 5.4. Numerous other functionals can be implemented by changing the configuration.

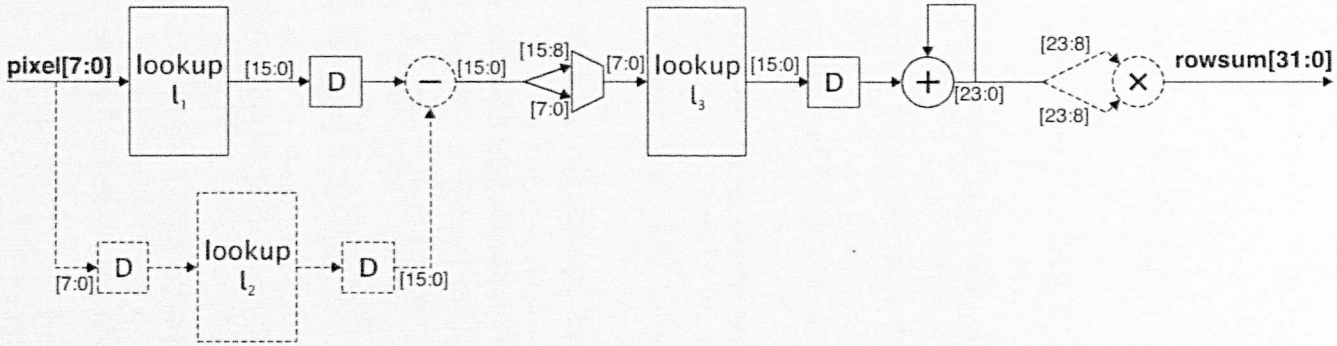


Figure 5.2: Type A functional block dataflow diagram. Implements functionals 1, 2 and 4 from Table 5.2.

Func. No.	Equation	l_1	l_2	l_3	Conf. Reg.
1	$\int_0^\infty f(t)dt$	x	–	–	- - - - 0000
2	$\left[\int_0^\infty \lvert f(t)\rvert^{\frac{1}{2}}dt\right]^2$	$\sqrt{x}$	–	–	- - - - 0001
4	$\int_0^\infty \lvert f(t)'\rvert dt$	x	x	–	- - - - 1001

Table 5.4: Type A functional configurations.

5.4.2 Type B Functional Block

This functional block is more complex than Type A; it can implement functionals 9, 11, 12, 13 and 14 from Table 5.2, which depend on the weighted median calculation. The weighted median is implemented using the efficient hardware architecture detailed in Chapter 6, with some modifications. Firstly, the sample and weight wordlengths are both set to 8 bits, as required for this block. The median block must also take into account the mask, so this is incorporated within the median calculation. Finally, there is no need for the FIFO block since this is not a sliding window implementation. Instead a reset signal is added to allow the bin counters to reset at the start of each row. Since the median filter block was implemented in VHDL, a wrapper was used within the Handel-C code. This wrapper tells the compilation tools to include the netlist for the VHDL object (which was separately generated using Synplicity Synplify) within the generated netlist for the Handel-C architecture. This merged netlist is then used within the Xilinx tools for mapping, placement and routing.

Since the median block can only return a result after a whole row has been processed, the current implementation uses the result from the previous row

Func. No.	Equation	l_1	l_2	l_3	Conf. Reg.
9	$\int_0^{\infty} rf(t)dt$	x	x	x	- - - - - - - 0
11	$\int_0^{\infty} r^2 f(t)dt$	x	x^2	x	- - - - - - - 0
12	$\int_{c^*}^{\infty} \sqrt{r} f(t)dt$	x	$\sqrt{x}$	x	- - - - - - - 1
13	$\int_{c^*}^{\infty} rf(t)dt$	x	x	x	- - - - - - - 1
14	$\int_{c^*}^{\infty} r^2 f(t)dt$	x	x^2	x	- - - - - - - 1

Table 5.5: Type B functional configurations.

in each calculation.

The functional computes the intermediate values c and r as described in row 9 of Table 5.2. The input l as shown in Figure 5.3 is the output of the pixel position counter mentioned in Section 5.2.1. $f(t)$ is a single-cycle delayed version of the pixel stream, in order to be aligned with the counter. Lookup l_1 offers flexibility in modifying the skew of the median calculation with the default being x as per the equations. Note that the value c is only updated once per row (shown shaded in Figure 5.3). Function l_2 is then applied to r and l_3 to $f(t)$. The top 12 bits of these values are then multiplied before being summed in an accumulator. A single bit in the configuration register determines whether to sum from zero or from c. Table 5.5 shows various configurations of the functional block. Others, outside of those shown in Table 5.2 can be implemented by changing the configuration.

5.4.3 Type C Functional Block

This functional block, shown in Figure 5.4, follows the structure of functionals 20, 21 and 22 in Table 5.2. The system follows similar design to the Type B

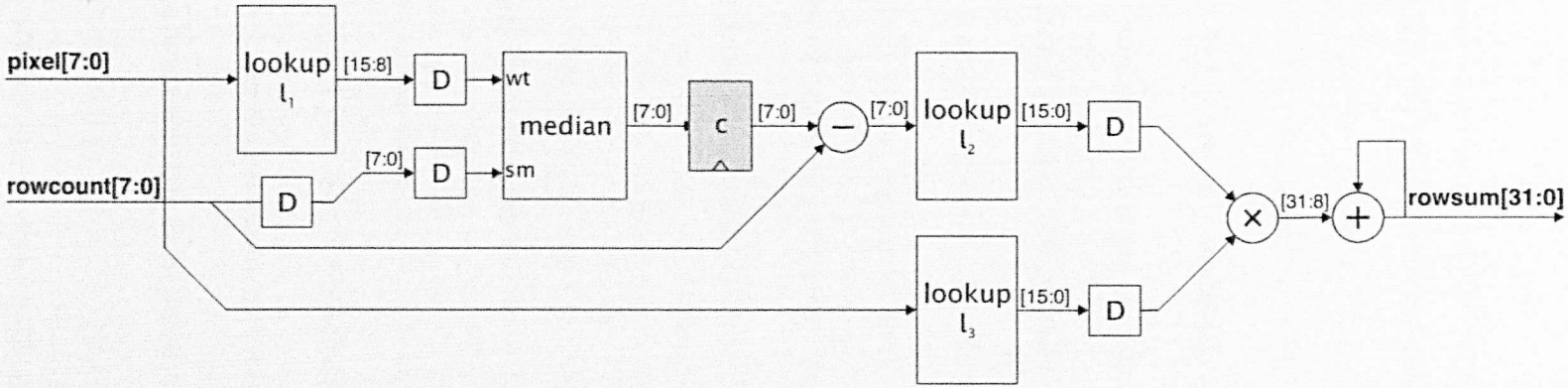


Figure 5.3: Type B functional block dataflow diagram. Implements functionals 9, 11, 12, 13 and 14 from Table 5.2.

Func. No.	Equation	l_1	l_2
20	$\int_c^\infty \sqrt{r} f(t)dt$	$\sqrt{x}$	x
21	$\int_c^\infty r f(t)dt$	x	x
22	$\int_c^\infty r^2 f(t)dt$	x^2	x

Table 5.6: Type C functional configurations.

functional block, except that c is computed as described in row 21 of Table 5.2. Note that the values c and S are only updated once per row. The division by S in the functionals is implemented using a lookup table, for simpler circuitry and pipelining. The input to this lookup table is truncated to restrict the number of embedded memories needed. To allow the full range of inputs, 64 Block RAMs would need to be combined into a 64K×18 bit memory. By truncating, a single Block RAM is sufficient. This functional block does not use a configuration register. Configurations for functionals 20, 21 and 22 are shown in Table 5.6

5.4.4 Functional Coverage

These three functional types cover 10 of the 11 functionals required by the Shape Trace Transform (STT) for face authentication [SPKK05]. Furthermore, by using alternative look-up functions in the Block RAMs and changing the values in the configuration registers, it is possible to add further functionals. The remaining functional used in the STT (Number 7 in Table 5.2) is a Fast Fourier Transform (FFT) that can be directly implemented using a predesigned core, with no need for flexibility.

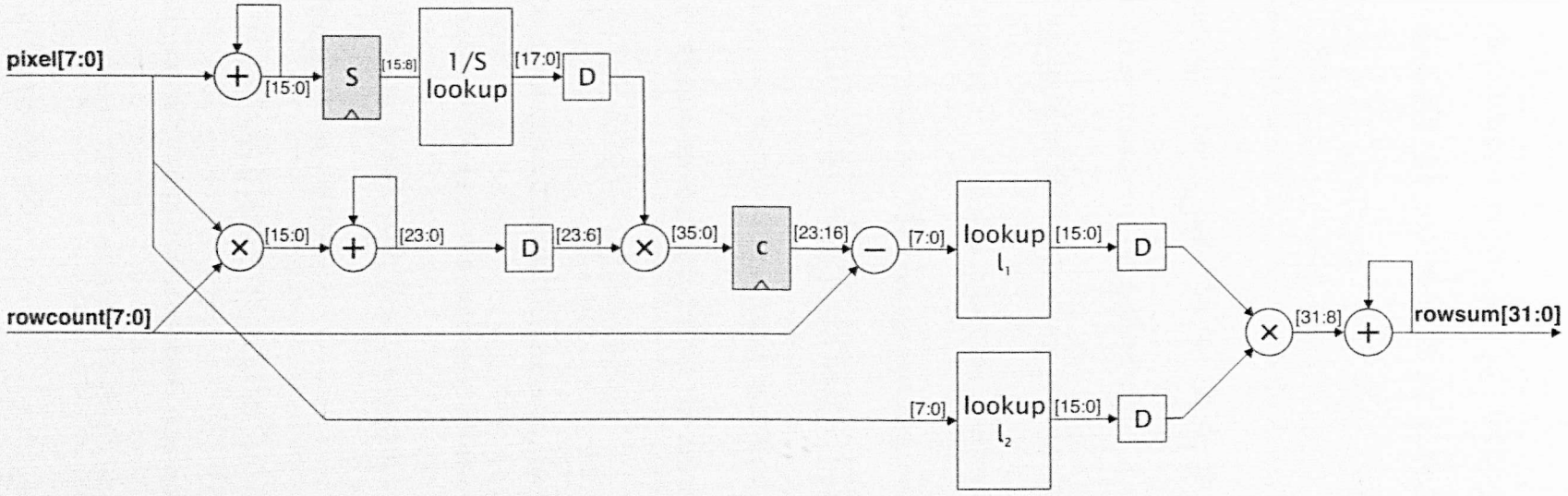


Figure 5.4: Type C functional block dataflow diagram. Implements functionals 20, 21 and 22 from Table 5.2.

5.4.5 Accuracy Considerations

In Section 5.2.2, the accuracy of the lookup functions was presented. Any inaccuracies can propagate through the system, so it is necessary to consider the overall accuracy for a functional. It is important to note that computing the Trace transform is not the same as applying another algorithm where the absolute values that result must be the same, having been defined precisely. With this Trace transform implementation, it would be used for both the training and recognition tasks in a given system. As such the requirement is that it is "self-accurate" – that it produces consistent results so that when multiple images are compared, the comparison can be made with a degree of confidence. The numerical data itself is not used in further mathematical processing that requires numbers to fit an exact specification. Indeed in the case of face verification, shapes are extracted from the trace through thresholding, thus discarding the fine accuracy of the numbers.

With the Type A functional block, the accuracy for the three defined functionals was found to be 100% for the functionals defined above. The inaccuracy of the square-root lookup is so small, that the maximum relative error for the whole of a trace image is only 0.02%.

For the Type B functional block, the use of the median result from the previous row in the calculation introduces some inaccuracy. The relative error has an average of 2%, with 52% of samples having zero relative error and 97% having less than 10% relative error.

For the Type C functional block, the use of old values of the c variable causes error, but with a different profile to that of the Type B functional block. In this case, the error has a more even spread, and there are a number of positions where the relative error is very large. 28% of pixels have zero error, while 86% have less than 10% relative error. 0.66% of positions have greater than 100% relative error, however.

It is important to note that these errors have little effect on the trace images when treated as images. The outliers are very few in overall terms and are spread sparsely at certain points where there is already high contrast. It depends largely on the next step in processing as to what constitutes acceptable error. Figure 5.5 shows traces computed using floating point arithmetic on the left and the equivalent computed in hardware on the right.

Note that the errors for the type A and B functional blocks, as a result of using one-row delayed intermediate can be overcome by adding a single full line latency to the whole system. While an additional 256 cycles of latency may at first seem significant, recall (from Section 4.6) that a complete trace takes over 3 million cycles to produce, and so it is a relatively small delay, and will not affect the throughput in any way. There is no other way of overcoming this accuracy issue, since these functionals produce results that depend on some property of the whole row being computed in advance. In this case, the errors were found to be tolerable for the types of processing expected to follow, and so the extra latency was not added.

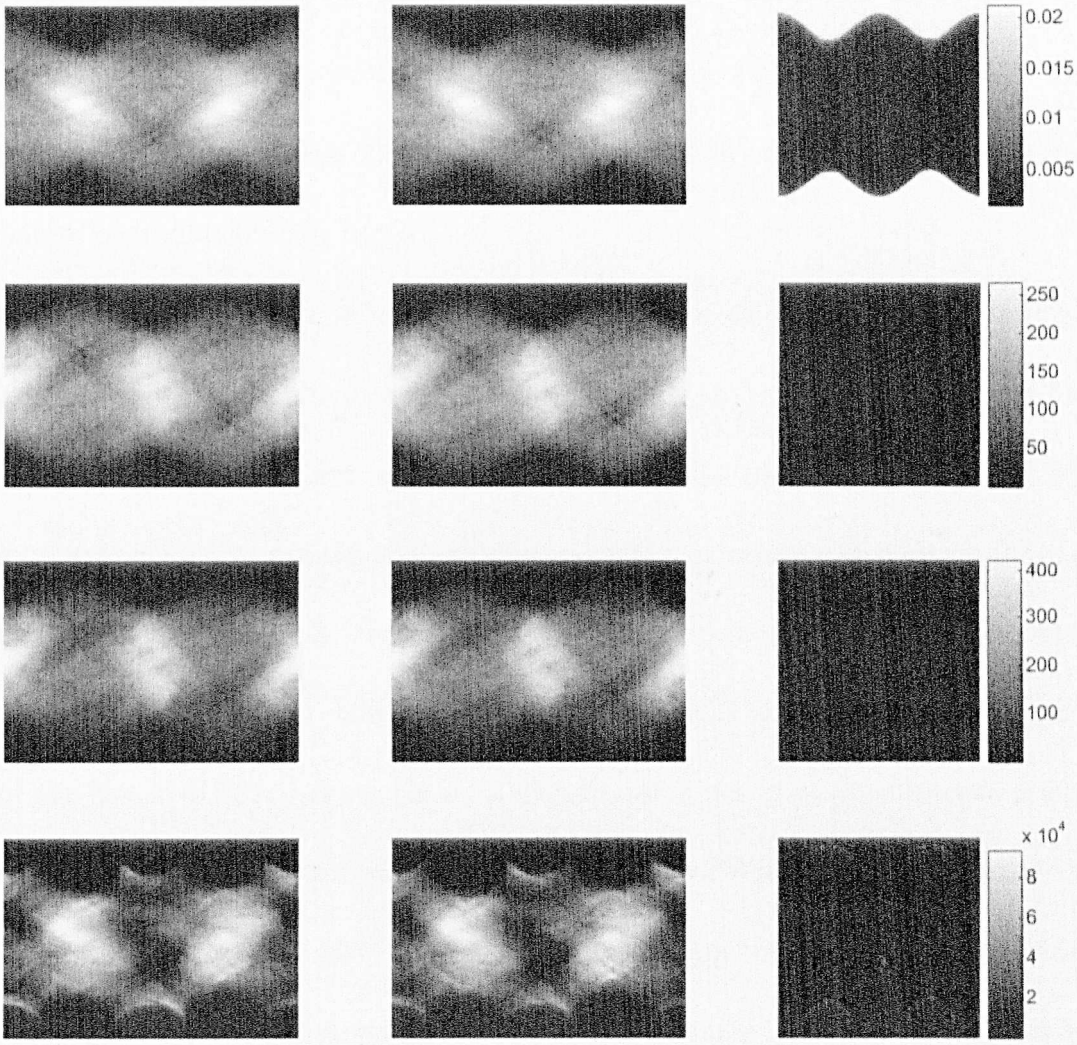


Figure 5.5: Trace images obtained using floating point arithmetic (left), the equivalents using the hardware architecture (centre), and error images with percentage range (right).

5.5 Initialisation

Having covered the design of flexible functionals and given examples, it is necessary to consider how the configurations are defined and applied. The flexible functionals rely on configuration data in order to function. The correct setting must be placed into the configuration register, along with the appropriate data for the lookup memories. This results in the addition of an initialisation stage before computation can begin.

Since there may be 8 to 10 functionals on the chip at one time, a large number of Block RAMs must be initialised. Doing so from one central location would be inefficient since signals would have to be routed to more than 40 locations from the initialisation block (assuming an average of 6 Block RAMs per functional block). In order to make this process more efficient, the system reads the initialisation values from the PC via USB and stores them in one of the board RAMs. From there, a distributor block reads the initialisation data and writes them to a shared bus as they are. Within each functional, a small block interfaces with this shared bus ignoring instructions that do not apply to it, and only activating when the functional is a match. When this is a case, a counter is reset and enabled, and this serves as the address input to the Block RAMs. The lookup number is used to select which Block RAM to write to, and the data values are read directly from the bus and written to the relevant location within the Block RAM. The system is shown in Figure 5.6.

The initialisation data are formatted as shown in Figure 5.7, with each

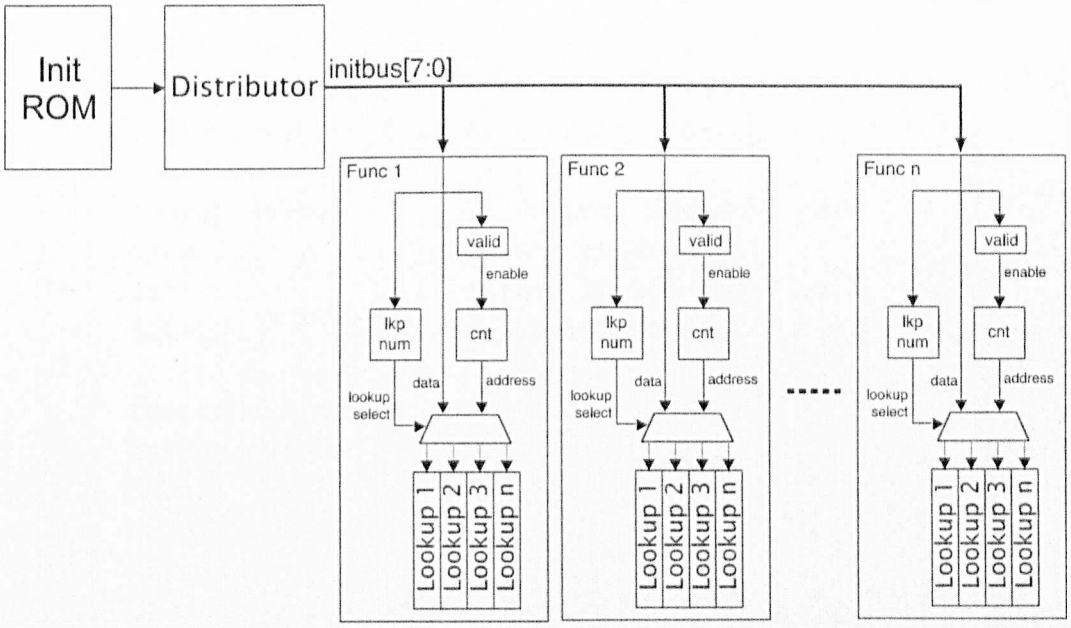


Figure 5.6: Flexible functional initialisation bus.

line representing a single byte of data read from the PC and stored to the board RAM. Since the data is transmitted over USB, any data longer than a byte must be split up. The first three bytes, when concatenated, determine the total length of the initialisation data. Following this, sets of instructions arrive, each headed by the functional block number followed by the lookup memory number. Then the contents of the memory are sent in order, split into bytes. Since the size of the lookup words is fixed at 16-bits, it takes 512 cycles to fill a BlockRAM. In order to initialise each configuration register, a reserved lookup number of 255 is sent followed by a single byte containing the data.

Following this modular approach means that the initialisation data can be in any order, and that the functional blocks can contain a variable number of lookup memories. The initialisation process only needs to occur at the

```
initialisation data size (byte 1 of 3)
initialisation data size (byte 2 of 3)
initialisation data size (byte 3 of 3)
functional number
lookup number     // 255=config register code
data 1            // Lookup data to be
data 2            // stored in the memories
data 3            // split into bytes.
...
functional number
lookup number
data 1
data 2
data 3
...
```

Figure 5.7: Functional lookup initialisation data format, as stored in the board RAM.

beginning of a processing run. The time taken depends on the number of Block RAMs that need to be initialised with each requiring 514 cycles per Block RAM and 3 cycles for each configuration register. At 80MHz, this translates to just over 640ns per Block RAM.

5.6 Performance and Area Results

The system implemented is identical to that in Chapter 4, but with these three flexible functionals. The system-level timing is identical to that shown in Figure 4.8. The only difference is that before the system begins processing, it waits for the completion of the initialisation phase detailed above. The system was synthesised with each of the above functional blocks in order to obtain separate area results for each. The resource usage for each of the functionals

Unit	Slices	Embedded Mult's	Block RAMs
Framework	1,300	4	2
Type A Functional	800	4	6
Type B Functional	23,500	4	22
Type C Functional	1,300	8	6
Total Available	33,792	144	144

Table 5.7: Synthesis Results for the Three Flexible Functional Blocks.

and the framework architecture are shown in Table 5.7.

All units were successfully synthesised to run at 79MHz. This limitation is enforced by the board libraries that are used to access resources on the board. This is the same speed as the architecture in Chapter 4, and shows that by pipelining the design, the complexity of the functionals does not affect the speed. While the timing allows for 36 functionals to be implemented, as previously mentioned, the area of each of the blocks must be taken into account. The actual number of functional blocks that can be implemented would depend upon the area requirements of each type and the resources available on the target FPGA.

Table 5.8 shows the speedup in computing a single functional using the discussed hardware implementation. As a software reference an optimised MATLAB version is used that is running on a Pentium 4 at 2.2GHz with 1GB of memory. The MATLAB version was implemented with nearest-neighbour rotations achieved using matrix multiplication for speed. It was coded making full use of MATLAB's vector operations and avoiding the use of loops. Just as with the hardware, lookup tables were used for computing the trigonometric

functions. Using the MATLAB compiler [Mat] yielded very minimal gains in performance, so it was not incorporated in these figures. While one might argue that the absolute performance gain may not be representative of a highly optimised software implementation, the important observation is that the performance of the hardware functional blocks is not affected by the complexity of a functional, whereas software reflects the complexity of a functional in its runtime.

In all the cases, the hardware design outperforms the software version by a considerable margin. It is important to note that these numbers are for a single functional. In the hardware implementation, additional functionals are computed in parallel, resulting in an even greater performance boost. Consider a system with three functionals - one of each type. The software implementation would take 7.9 seconds to compute the three traces, whereas the hardware system would still take 38.5 ms, a speedup of over 200 times. Increasing the number of functionals further, to the 11 implemented out of Table 5.2, a software version takes 31 seconds, while the hardware implementation would still take the same time as implementing one functional. This results in a speedup of over 800 times. This would require a larger FPGA device since the area requirements are significantly more than is available on the current target platform, however higher capacities are common in the latest generation of FPGAs [Xil07b, Alt07].

It would be possible to compute the 11 functionals described using just the three blocks presented here. In such a case, the system would need to

Functional Type	S/W (ms)	H/W (ms)	Speedup
Type A Functional	1400	38.5	36x
Type B Functional	3900	38.5	101x
Type C Functional	2600	38.5	67x

Table 5.8: Running times and speedup factors.

run through the trace computation 5 times, to allow the Type B functional block to compute all its variations. This would result in a total runtime of $38.5 \times 5 = 192.5$ms, a speedup of over 160× over software.

The number of functionals that can be implemented in hardware is limited by two factors. Firstly, the resource requirements of a functional block and the resources available on the target device must be taken into account. Clearly, the number of functionals possible in an implementation depends entirely on the combination of functional block types required.

The second factor is timing-related. A full trace computation takes just under 3 million cycles to complete, and since the input and output memories are double-buffered, it is necessary for the data transfer to and from the board to be completed in this time. The loading of input data from the PC takes 327,860 clock cycles as detailed in Section 4.4.3. Each functional produces a trace image that is 256×256 pixels in size[2], with each pixel being 32 bits wide. Hence reading a trace image in bytes over USB takes $256 \times 256 \times 4 = 262,144$ cycles. This means that the maximum number of functionals that can be accommodated in this implementation is $\lfloor (3\text{M} - 327,680)/262,144 = 10 \rfloor$. This

[2]The actual result is a 256×180 image, but for ease of address calculation, it is stored within within a 256×256 size memory block.

limitation is due to the use of the USB port to transfer data. An FPGA board with a wider and/or faster interface to the PC would alleviate this problem.

5.7 Summary

In this chapter, a framework for designing flexible functionals for the Trace transform was presented. The main design consideration is to introduce flexibility into the functional blocks, allowing a single block to compute a number of different functionals. Employing Block RAMs as lookups for function evaluation leads to a highly pipelined system with significant flexibility. A flexible, efficient initialisation scheme was also presented. Three example blocks based on functionals used for face authentication [SPKK05] were developed using this scheme and shown to perform as fast as the fixed blocks shown in Chapter 4. By instantiating multiple functional blocks, the system achieves over 2 orders of magnitude acceleration over a software implementation, with this factor increasing for the more complex functionals.

The framework presented can be applied to develop further flexible functional blocks that can aid significantly in functional exploration for different applications of the Trace transform. This can result in a much better choice of functionals for a given application, rather than relying on a library that may serve well for some applications while not performing well for an application at hand. This opens the door to investigating the use of the Trace transform in a wide variety of application domains beside those already investigated.

Chapter 6

Large-Windowed, One-Dimensional Median and Weighted Median Filters

6.1 Introduction

One of the mathematical functions that is used extensively in Trace transform functionals is the median and weighted median [SPKK03, SPKK05]. Figure 6.1 shows where the Trace transform fits into the Trace transform implementation.

The median of a set of samples is often computed by sorting the input samples then selecting the middle value. The weighted median can be computed in multiple stages: first expanding the weighted sample sequence, then sorting and finally locating the median. However, these methods are not suitable as a block in a fully pipelined system, since results are not produced immediately

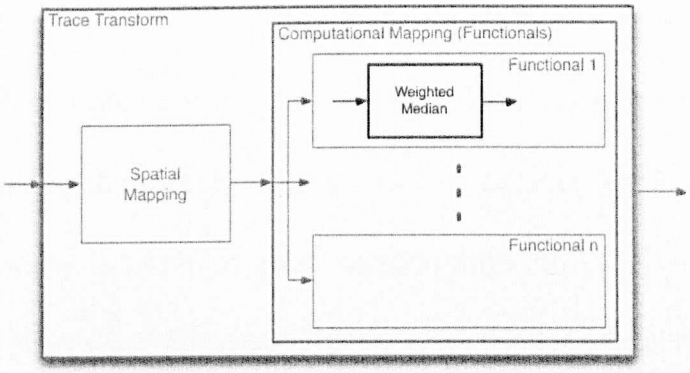


Figure 6.1: Position of median and weighted median circuit within a Trace transform implementation.

but after multiple stages. This requires intermediate storage and complex control logic to pause the previous stages in a circuit. Furthermore, the median and weighted median calculations used for the Trace transform are typically on large windows, equivalent to the length of a line crossing an image ($\tilde{2}00$ samples). Also, in the case of the Trace transform, the length of the sample window is variable. These two facts present a problem for the standard implementations of median and weighted median filters. In this chapter, a real-time hardware architecture is presented that can compute the median and weighted median over a flexibly sized window. The architecture is fully pipeline-able and thus is easy to incorporate into a pipelined system such as that developed in Chapter 4. Part of the work in this chapter was published in [FCL05b].

6.2 Definition

The median filter is a highly versatile non-linear filter that has been used extensively in a variety of domains. Its strength lies in its ability to filter

out noise while minimally affecting the properties of the underlying signal. The median filter replaces a sample with the middle ranked value amongst all the samples within the sample window. In this manner, it filters out samples that are not representative of their surroundings, in other words, outliers. In the image processing domain, a two-dimensional median filter allows for the removal of "salt-and-pepper" type noise from an image without adversely affecting the underlying edges. The use of a linear filter (such as a Gaussian or mean filter) in this situation would cause a blurring of edges. The median filter can still degrade image quality somewhat, though the preservation of edges is paramount in the computer vision domain.

Given an input sequence $x_1, x_2, x_3, \cdots$, a window of size $2K+1$ is defined, centred on the ith value as $W_i = \{x_{i-K}, x_{i-K+1}, \cdots, x_i, \cdots, x_{i+K-1}, x_{i+K}\}$. The output of the median filter, y_i, is thus the median of W_i; the middle value in the sorted list.

The weighted median is an extension of the standard median, wherein each input sample also has an associated weight that determines how much that sample contributes to the final result. Weights can have fractional or integer values. From a computational perspective, this makes no difference as long as fractional weights are fixed point and lie within the same limits for all samples. Negative weights are undefined.

The input sequence for a weighted median filter can be written:

$$(x_1, w_1), (x_2, w_2), (x_3, w_3), \cdots,$$

where x_i are the input samples and w_i are the corresponding weights. An integer weight would simply correspond to having w_i copies of sample x_i taken into account in the median calculation. Consider the example sequence:

$$(1,3),(5,1),(2,5),(4,2),(7,2),(3,5),(4,1).$$

After expanding, this becomes:

$$1,1,1,5,2,2,2,2,2,4,4,7,7,3,3,3,3,3,4.$$

To determine the median, this sequence must be sorted as follows:

$$1,1,1,2,2,2,2,2,3,\mathbf{3},3,3,3,4,4,4,5,7,7.$$

Finally, the middle value in the sequence, 3, is selected as the weighted median of this series.

It is important to note that for the weighted median, the size of the window is the sum of weights rather than the number of tuples received. So for the above sequence it is 19 and not 7. The median index can be calculated by halving this number and adding one, so in this case the median index is 10.

Much of the literature dealing with median filters in image processing is focused on 2-dimensional filters of small size [Ric90]. Weighted median does not have a widespread use in image processing and thus little work has been done on efficient implementations. The number of sample points required in

the Trace transform can be very large, in the order of hundreds. Furthermore, the window size, in the case of the Trace transform, is not fixed, so an implementation must be flexible in this regard. Designing an implementation that can facilitate such calculations over large windows would allow for a full hardware implementation of many Trace transform functionals.

6.3 Related Work

Median filters have been implemented in hardware in a variety of ways. [Ric90] provides a very good review of the area. There are two main methods, the first is to maintain the input sample list in its original order, then pass it through some type of sorting network. The median value is then extracted from the relevant position in the ordered list. The other method involves sorting the samples as they enter the system. Of the first approach, the simplest implementation is the bubble sorting grid, where a grid of dual input sorters each swap their inputs to propagate the higher valued samples upwards, and lower valued samples downwards (or vice-versa). The median is simply the middle sample of the grid output. An example of this architecture is shown in Figure 6.2. This method is regular yet its hardware requirements increase in proportion to the square of the window size and hence it is not scalable to larger windows. For a window of size $2N + 1$, $N(2N + 1)$ dual input sorters and $2N + 1$ registers are required as can be seen in Figure 6.2.

For small windows, simplifications can be made [BN97], where the columns,

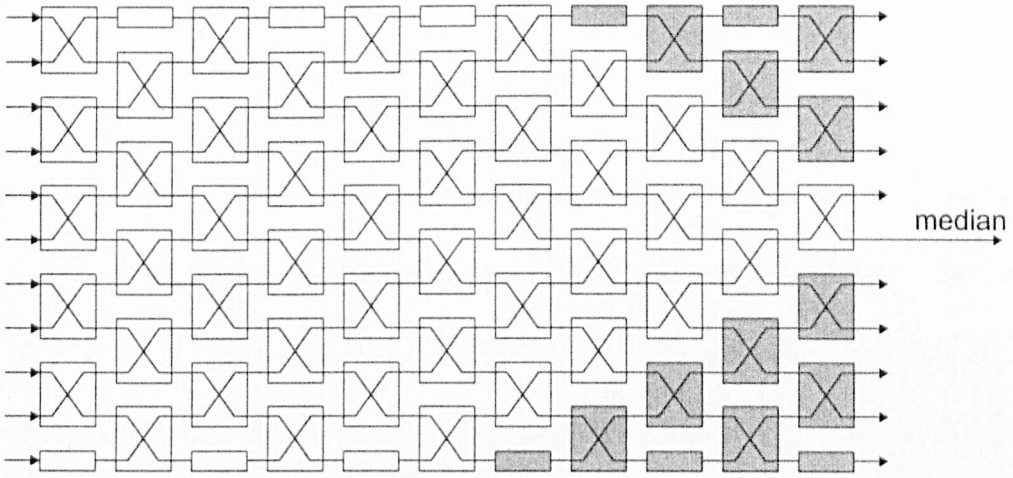


Figure 6.2: A simple 11-sample bubble-sorting circuit layout. The large blocks are compare-swap units that swap their inputs if necessary to propagate the larger values upwards and the smaller ones downwards. The small blocks are registers. Note that the shaded blocks are not required for median calculation.

then rows of a 2-dimensional window are each sorted using a triple-input sorter. Then only one diagonal needs to be sorted to give the median. This saves on hardware requirements. Karaman et al. [KOA90] propose a change to the standard sorting network by dealing with samples in a bitwise manner, needing only single bit sorters, however their implementation is still proportional to N^2 in area. The strength of regular array architectures is that they can be pipelined down to a single compare-swap stage. This results in high throughput. Benkrid and Crookes [BCB02] create a sorting structure based on Quick Sort using a bit-voter block; the area requirements are $O(N)$. Other methods that use fewer building blocks of higher complexity are described in [YLC99, CCH96, BP02]. Another method is that of threshold decomposition, as used in [BT04], however the architecture proposed relies on the window being of size 3×3 and uses 3-input adders and so is not scalable to large windows. Systolic median architectures based on insertion sort have also been proposed [GL01]; in this case,

the amount of hardware is proportional to the window size. In [VRSPGP02], the authors take advantage of the wide data buses on the development board to allow the median calculations for multiple pixels in parallel. The overlapping data between 3×3 windows is re-used and the sorting circuit is modified to reduce the number of compare-swap blocks. The proposed architecture is, however, limited to 2-dimensional windows of 3×3 pixels and larger windows would not scale due to the sorting circuitry.

Another method for computation of the median of a sequence of numbers involves computing the cumulative histogram for this sequence, then finding the index of the first bin total to exceed the median index. The principle is well established and known, having been mentioned in basic textbooks on image processing. [AP94] and [HFC95] both deal with software implementations of this algorithm running on general-purpose processors. Presented here is the first implementation and analysis in hardware of the proposed method. The high degree of parallelism that can be had in hardware, coupled with the independence of the area with regard to window size, is what makes this method so attractive as compared to a sorting structure. Furthermore, this method is extensible to the weighted median as will become apparent.

6.4 Proposed Architecture

6.4.1 General Overview

The proposed architecture works by constructing a cumulative histogram of the input data. This is done by maintaining a count for occurrences of each possible input value. Since the application domain in this case is video processing, 8-bit unsigned numbers (let $l = 8$) have been assumed. This means there are $2^8 = 256$ possible input values, and so a rank of 256 bins is used. To construct a histogram, when an input value is received, the bin corresponding to the sample value is incremented. For a cumulative histogram, each subsequent bin must also be incremented. In software, this is normally done as an additional step after the histogram has been fully populated. A pass through all the bins adds the value of the previous bin to each bin. Hence, the value stored in the final bin will always be equal to the number of input samples received. The median is then simply the first bin whose count reaches or exceeds the median index.

For example, if the median is to be calculated over a window of 101 elements, i.e. $2K+1 = 101$, $K = 50$, then the 51st or generally $(K+1)$th element in the ordered list is required. Using the histogram, find the first bin whose count is 51 or above, this gives the median of the input samples, since the 51st ordered element must lie in this bin.

To implement this in hardware, a rank of parallel bins is instantiated. The count value for each bin is compared to the median index (in this example,

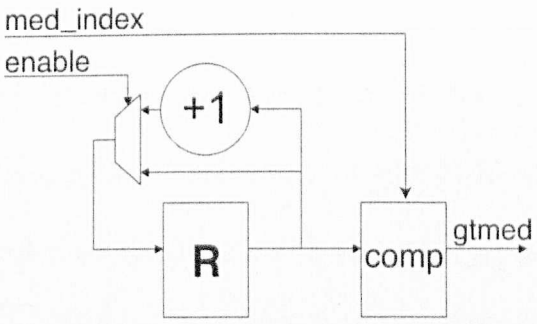


Figure 6.3: A histogram bin node processor.

51), resulting in a 0 if the bin count is smaller, and a 1 if it is equal or larger. Hence the result for all bins before the one containing the median will be 0, and all the others will be 1. A priority encoder can then be used to find the index of the first bin in the series of 1's. A priority encoder takes an B-bit input in which there are b zeros followed by $B - b$ ones, and returns b. This gives the median of the input.

A separate register is required to keep a count for each of the possible input values. Hence 256 registers are needed to store the counts for 8-bit samples. For the registers to all be updated in parallel, each register also requires its own incrementer, which is activated only when that bin needs to be incremented. (Recall that to construct a histogram, only the bin with an index corresponding to the input sample needs to be incremented.) Hence, each bin has an enable input that determine whether it should be incremented in the current clock cycle, and the median index as an input. The output is a single binary value that is 1 when the value equals or exceeds the median index and 0 otherwise. This gives the design for a bin node as shown in Figure 6.3. 256, or in the general case of l-bit samples, 2^l, such nodes are required in

the proposed system. Note that the width of the bin registers depends on the window size required. This will be investigated further in Section 6.5.2.

A circuit composed of such processors would yield the histogram of the input signals. In order to compute the cumulative histogram, some further processing is needed. As mentioned above, it is possible to separate the construction of the cumulative histogram and do this as a subsequent step. This, however, would be wasteful, as the accumulation for each bin would have to be done in turn, taking 256 cycles in total. One possible alternative approach is to instantiate a comparator for each bin, and compare the input sample value to the index of each bin. Those bins with an index greater than or equal to the input sample value would be incremented. However, this would be costly in terms of hardware, since each bin would require its own l-bit comparator.

Another approach would be to connect each bin to the previous one, such that if the previous bin is being incremented, then it would increment too. However this would slow the system down significantly, since that incrementation signal would need to propagate through 256 stages in the worst case, all in one clock cycle. Analogous to this is the carry chain in a carry-ripple adder.

A more efficient method, that takes advantage of the heterogeneous resources on modern FPGAs, is to use embedded Block RAMs on the FPGA as a ROM to store the bin access patterns. For the 8-bit inputs previously mentioned, a 256×256-bit ROM would be required to decode the 8-bit number to a 256-bit signal, where each bit represents the select input shown in Figure 6.3, to the corresponding bin; each bit of the output addresses a single

Address	Contents[0:255]
0	0xFFFFFFF...FFFFF
1	0x7FFFFFF...FFFFF
2	0x3FFFFFF...FFFFF
3	0x1FFFFFF...FFFFF
4	0x0FFFFFF...FFFFF
5	0x07FFFFF...FFFFF
⋮	⋮
253	0x0000000...00007
254	0x0000000...00003
255	0x0000000...00001

Table 6.1: Access pattern ROM contents.

bin node processor. The access patterns stored in the ROM, ensure that the correct bins are enabled for any given input sample. The contents of the ROM are shown in Table 6.1, while an overview of the circuit is shown in 6.4

This method of constructing a cumulative histogram is highly efficient and allows for a fully updated histogram in every cycle. This method has also subsequently been adapted for histogram equalisation on images [AA05][1], and shown to perform significantly better than a software implementation on a Graphics Processing Unit (GPU) [CCL07]. Note that histogram generation is just one part of the median and weighted-median implementation.

[1]Note that the cited paper does not reference this work as published in [FCL05b]. [AA05] was published in December 2005, having been initially submitted in July 2005. [FCL05b] was submitted in March 2005, accepted in May 2005 and presented/published in August 2005.

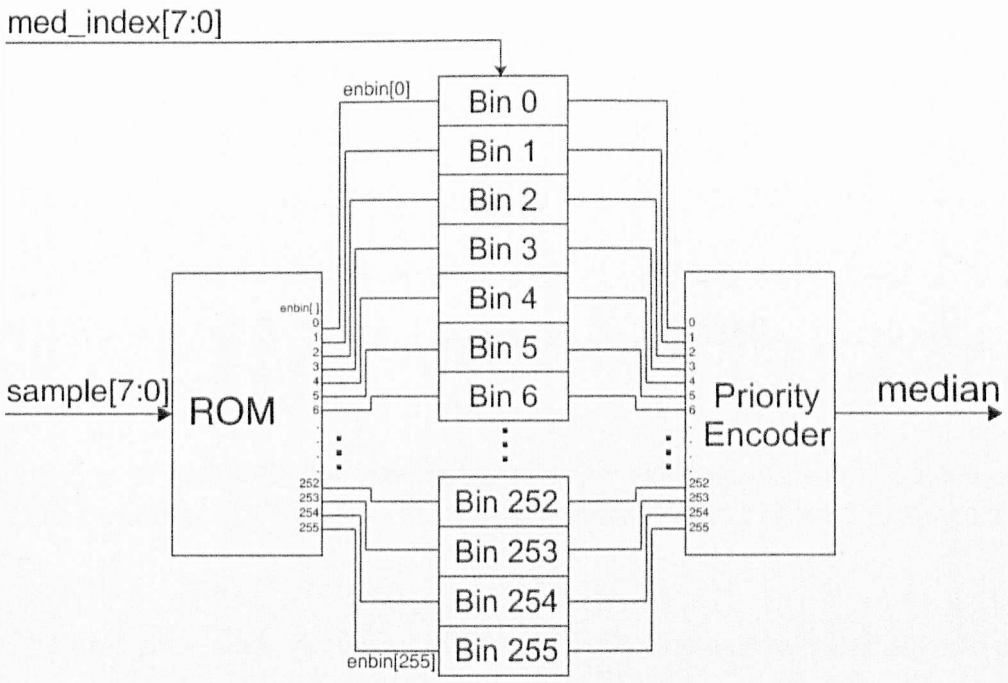


Figure 6.4: Histogram-based median filter architecture.

6.4.2 Sliding Window Implementation

Thus far, the system takes a sequence of samples and returns the median up to each point. Such a circuit, however, is not useful, since normally, the median must be computed for a fixed-size window of values. Often, the filter is implemented as a sliding-window. This means that in each cycle, the window moves one sample down the sequence, discarding the oldest sample and adding the newest into the window. To implement this algorithm for sliding windows, some changes must be made. Consider that now while constructing a histogram, with each new sample that enters, the oldest sample is removed from the window, and thus its effect on the histogram must also be negated. This however only happens after the window has become full. Hence some way of keeping track of the old samples, knowing when the window has become full

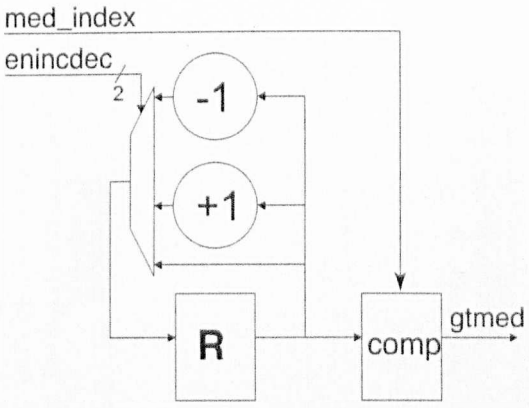


Figure 6.5: A bin node for the sliding window implementation. enincdec determines whether the count is incremented, decremented or kept at its current value.

for the first time, and some way of updating the histogram based on the new and oldest samples must be devised.

Firstly, a FIFO buffer is used to store the samples for the window over which the median must be found. When a new sample is received and the window is full, the oldest sample is removed from the FIFO. Updating the histogram requires all bins corresponding to the access pattern for the oldest sample to be decremented. At the same time, the bins corresponding to the new input sample must be incremented. This can all be done in one cycle, by simply leaving any bins that are included in both sets unchanged, since they increment and decrement at the same time. Bins that are only enabled by the access pattern of the new sample are incremented, while bins enabled only by the access pattern of the removed sample are decremented. Updating the histogram in this fashion means that it is up to date in every clock cycle, and there need not be a pause in the input samples. The new node design is shown in Figure 6.5.

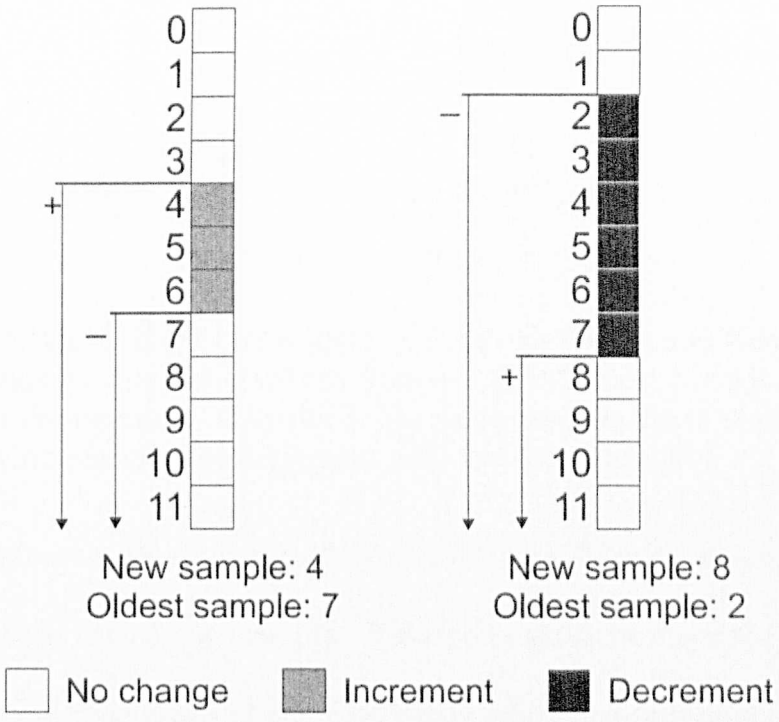


Figure 6.6: Application to sliding windows. The arrows aside the bins show the access patterns for the oldest (-) and new (+) samples. The leftmost example shows a new sample value of 4 arriving while the oldest sample is of value 7. Only bins 4 to 7 need to be incremented, all others keep their current values. The rightmost example shows a new sample of value 8 arriving, while the oldest sample is of value 2. Only bins 2 to 8 need to be decremented; others are left alone.

On-chip Block-RAMs are particularly useful for this architecture. Since these RAMs are dual-ported on the target architecture, it is possible to extract the enable signals for both the new and oldest samples from the access pattern ROM in parallel. These can then be processed to determine which bin is incremented. This is illustrated in Figure 6.6.

To implement this, a simple 2-input, 2-output lookup-table is required to determine the resultant action. This is shown in Table 6.2. This small logic function must be implemented for each bin. Recall that as the window is filling with values the first time, no subtractions take place, since this would mean

OldEn	NewEn	enincdec[1:0]
0	0	00
0	1	10
1	0	01
1	1	00

Table 6.2: Extra sliding window logic. The signals OldEn and NewEn are the enable signals for the bin resulting from the ROM lookup of the oldest and new samples respectively. enincdec is the signal that instructs the bin counter whether to increment (10), decrement (01) or do nothing (00).

that the histogram would never fill up with values. As such, a single valid bit is appended to each input sample. This propagates through the FIFO and emerges at the final stage of the FIFO only when one full window of values has been received. This bit is ANDed with the bin subtraction control signal, so no subtraction can take place until it emerges. The revised architecture is shown in Figure 6.7.

6.4.3 Extension to Weighted Median

To implement weighted median in the proposed architecture, further changes to the architecture in Figure 6.7 are needed. Recall that the weighted median is computed on samples that have associated weights and that those weights are equivalent to duplicating the sample the corresponding number of times. Further recall that the window size, and thus median index is dependent on these weights. To construct the histogram for weighted samples, rather than increment each bin for corresponding samples, the weight of that sample is added to the corresponding bin. The cumulative histogram is constructed as

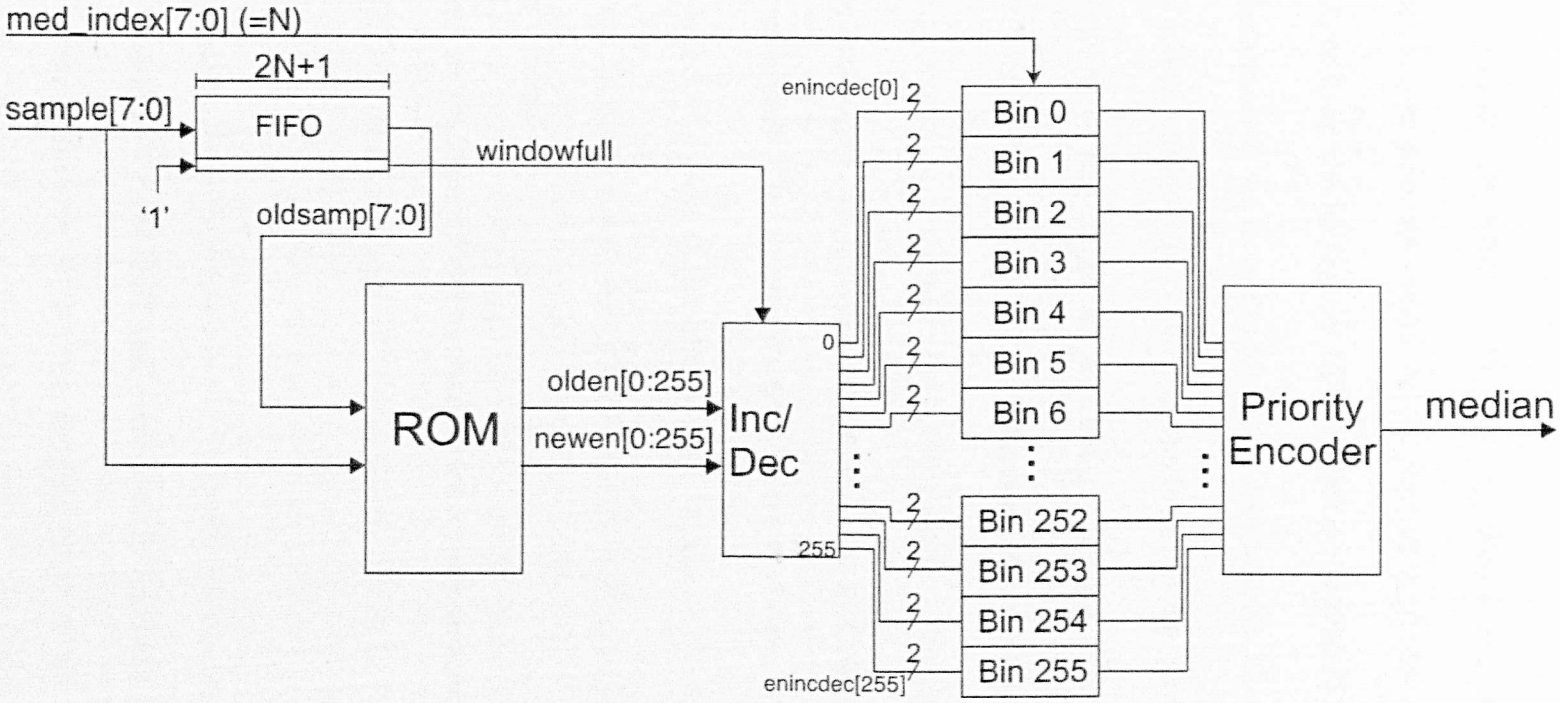


Figure 6.7: Architecture of the sliding window median filter.

per the standard median.

To make the necessary changes in hardware, another input signal is introduced to provide the weights. Instead of a simple incrementer, each bin processor must now add the weight value. Just as with the standard median, it is possible to keep the histogram fully updated at each clock cycle. Another FIFO is instantiated, to keep track of the old samples that fall out of the window. For bins enabled by the access pattern for the oldest sample falling outside the window, the weight of that sample is subtracted. For those bins enabled by both access patterns, the difference of the two weights is added (while being careful to maintain the correct sign). For those bins enabled only by the access pattern for the new sample, the new sample's corresponding weight is added. The resultant architecture is shown in Figure 6.8. The three signals fed into each of the bins are the weight of the new input sample, the difference in weights and the weight corresponding to the sample falling outside the window.

The rank, or position, of the median is not known in advance for weighted median. Consider the expansion of the sequence shown in Section 6.1, and it becomes clear that the number of 'real' samples received is equal to the sum of sample weights. Hence, the index of the median must be half of that plus one, which is simply a right shift and increment in hardware. In the proposed architecture, the difference of the two weights (that of the new sample and that of the oldest) is simply added to a register on each clock cycle. This maintains the current sum of weights. This is right shifted to divide by two and

incremented and fed into each of the bins, where it is used for the comparison.

The word-length of each bin register must be wide enough to accommodate the maximum sum of the weights to prevent overflow. In order to do this, the width of the bin counters must be equal to $\log_2$ of the window size plus the width of the weights.

6.5 Implementation Results

Implementation of the above designs was originally coded in Handel-C and compiled using the Celoxica DK Compiler. The target device in this case is a Xilinx Virtex II 6000, as found on the Celoxica RC300 development board. For comparison, an alternative implementation of the median filter based on the sorting grid mentioned in Section 6.3 was also synthesised.

Using Handel-C was found to give acceptable area and speed results for the sorting-grid architecture. However, due to the extra control logic that Handel-C inserts into a design, and the high level of parallelism in the proposed architecture, routing delays due to large fan-out of control signals was causing the circuit to have a high clock period. For the proposed architecture, the area usage was halved and the clock period reduced by over 60% when it was re-implemented in VHDL. The design was thus re-implemented and compiled using Synplicity Synplify Pro [Syn].

The reason for this disparity is a function of how Handel-C is implemented in hardware. A Handel-C circuit functions using token passing, effectively

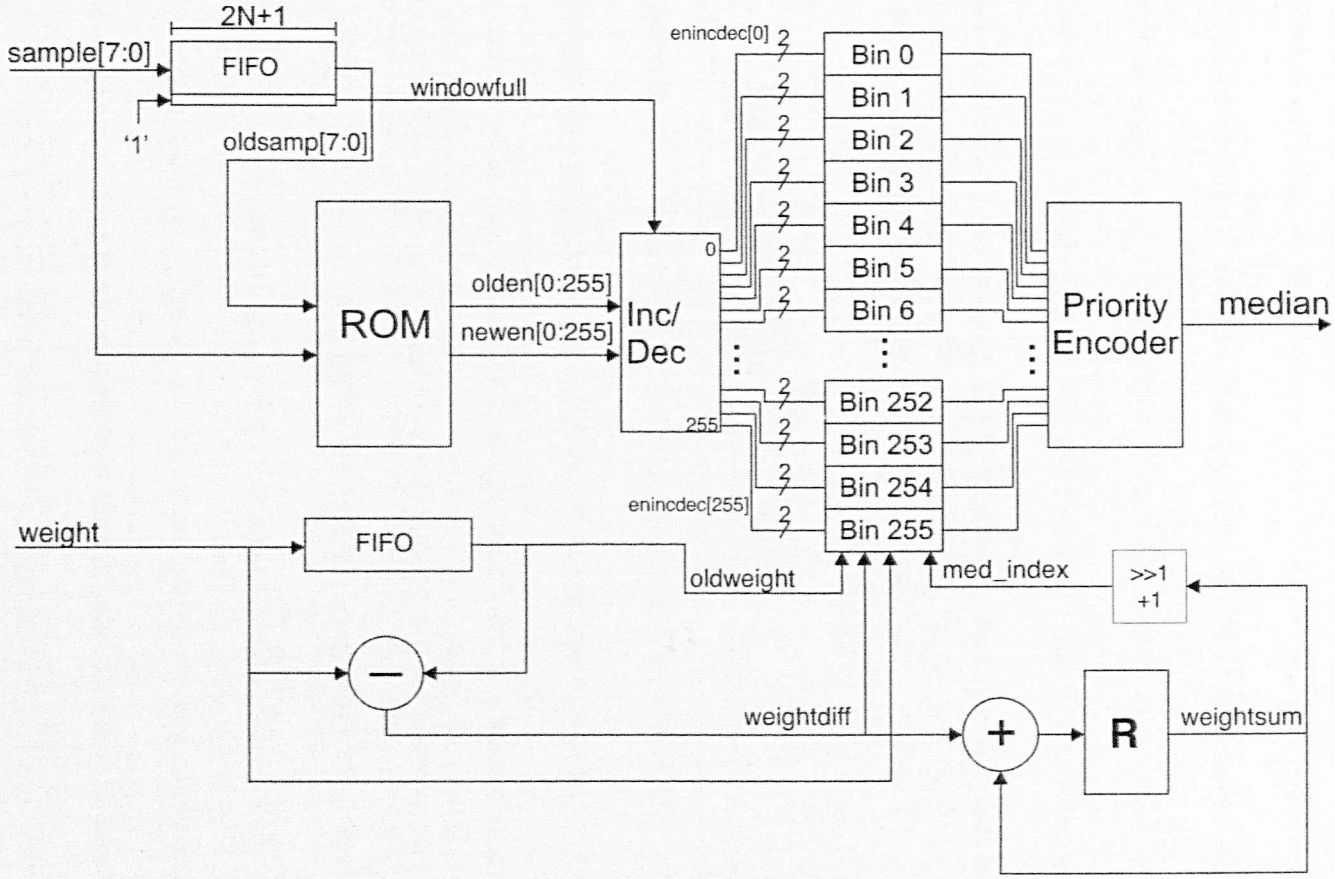


Figure 6.8: Architecture of the weighted median filter.

enabling subsequent parts of the circuit. Since each line of code takes a single clock cycle, this token is like the enable signal for all the instantiated hardware. The problem arises when there are a large number of units that have to be enabled in parallel, as with the 256 bins in this case. The single token passing signal must be fanned out to a huge number of circuit elements and this fanout introduces significant routing delay. Hence this problem is limited to massively parallel circuits. While Handel-C most definitely introduces some performance penalty, one must bear in mind the significant advantages afforded in describing complex circuits and using on-board resources.

6.5.1 Design Variations

In order to thoroughly investigate the proposed architecture, a number of variations were considered. Fixed window implementations were ignored, since they are of little use, returning a single result for a whole window. Instead, sliding window implementations were favoured due to their computation of a new result every cycle. A number of design parameters were varied, leading to multiple implementations. Before discussing these, consider the parameters that might affect area. Firstly, all implementations were synthesised for sample widths of 8-bits. This is an assumption that is valid for most of the calculations one would wish to conduct on images. Furthermore, this is the only significant limiting factor for this design due to the fact that the number of bins varies exponentially with the sample wordlength. Since this implementation is to be used in the Trace transform, 8-bit wordlengths are sufficient.

The counters in each of the bins need to be wide enough to accommodate the maximum count, equal to the maximum number of samples to be considered, which is equivalent to the window size. Hence the width of the bins is equal to the base-2 logarithm of the window size. One can set this arbitrarily to a fixed number such as 8-bits. This would allow for window sizes up to 255 samples. However, to keep the design as compact as possible it should be set to the appropriate width. The window size also affects the length of the FIFO buffer used to track older samples. This buffer is equal in length to the window size. Finally, one may choose to implement a design that uses a fixed window size, or one in which the window size is determined by the number of samples entering the system. The advantage of the second method is that the window size can be changed in runtime. The first method would synthesise a fixed value comparator. While this saves area, the window size must be fixed.

6.5.2 Synthesis Results

The first set of results, shown in Figure 6.9 shows the area usage for implementations and how this varies with the window size for each of three metrics: Look-Up Tables (LUTs), Flip-Flops (FFs) and Slices. These implementations were for fixed window sizes using hard-wired fixed value comparators. The vertical lines in the graph indicate the boundaries of different wordlengths for the bin counters.

It is clear from the graph that each time the counter wordlength requirements increases by one bit, there is a distinct jump in area requirements, in

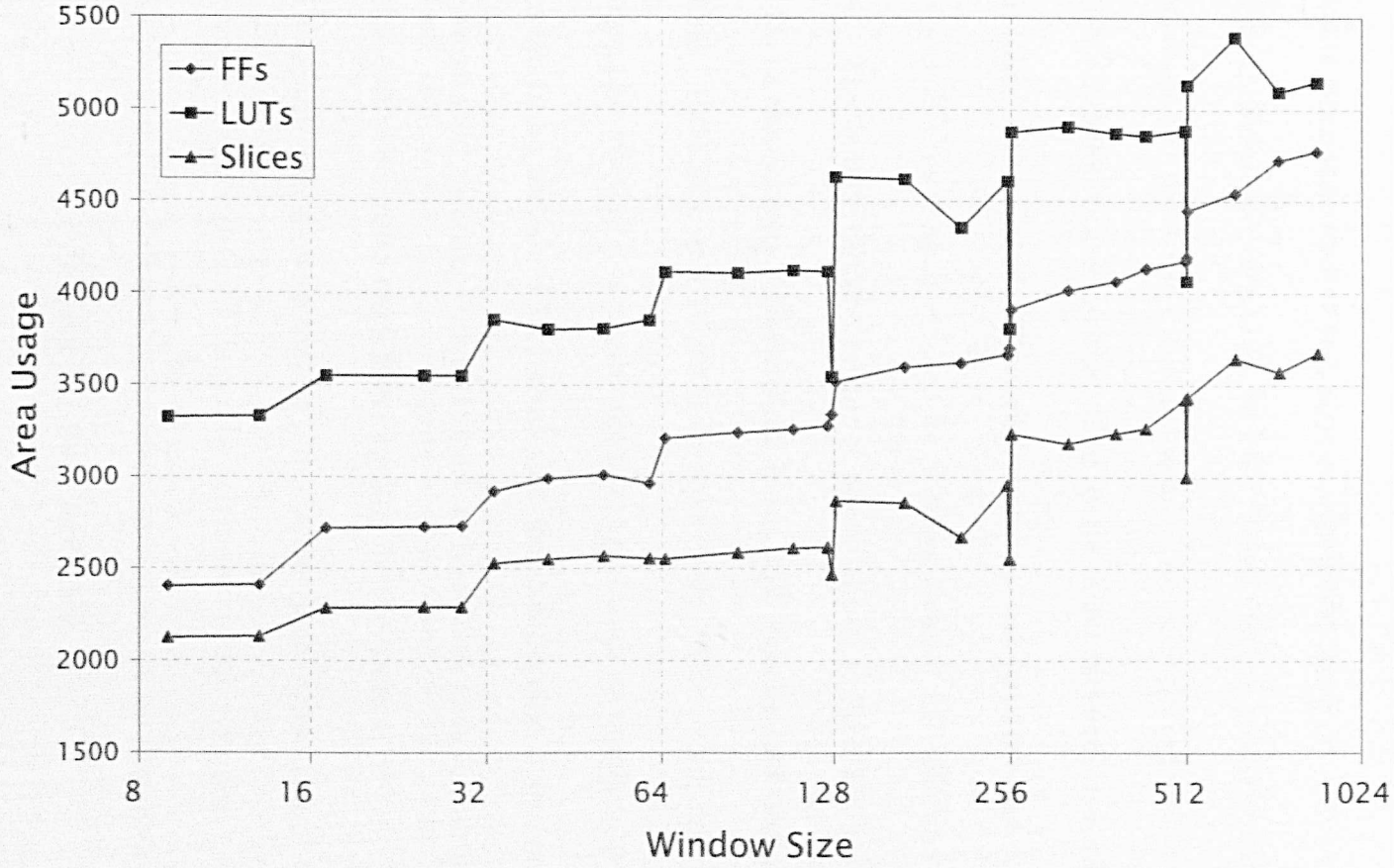


Figure 6.9: Graph of synthesis results for various window sizes.

the general case. The Flip Flop usage has a general rising trend, even between designs with counters of the same wordlength. This is due to the increasing size of the FIFO buffer; this also explains the increasing gradient of the FF segments since the window size is on a logarithmic scale.

The considerable variations in LUT usage can be put down to the optimisation of the fixed value comparator. When comparing values to a fixed number, and depending on the value of the fixed number, not all bits need to be taken into account. The synthesis tools will optimise the comparators as required. This is most evident for window sizes of 127, 255 and 511 in the graph. The binary representation of these values is 1111111, 11111111 and 111111111 respectively. The median index will thus be half plus one, giving 1000000 , 10000000, and 100000000 respectively. When comparing a number to determine whether it is greater than or equal to these numbers, only a single bit needs to be tested. This means that the comparator is reduced to a one bit comparator, resulting in a significant reduction in area. Other fluctuations are the result of similar reductions applied by the tools.

The graph also shows a lack of jump in the Slice count around the 64- and 512-sample window sizes. This can be attributed to the synthesis tools packing the LUTs and FFs differently, resulting in a more dense arrangement within the slices. Again, the designer has little input into this.

The general trend for area requirements can thus be described as being of the form $K + \log_2 N$, where N is the window size and K is the fixed area required by the rest of the design regardless of window size. The graph in

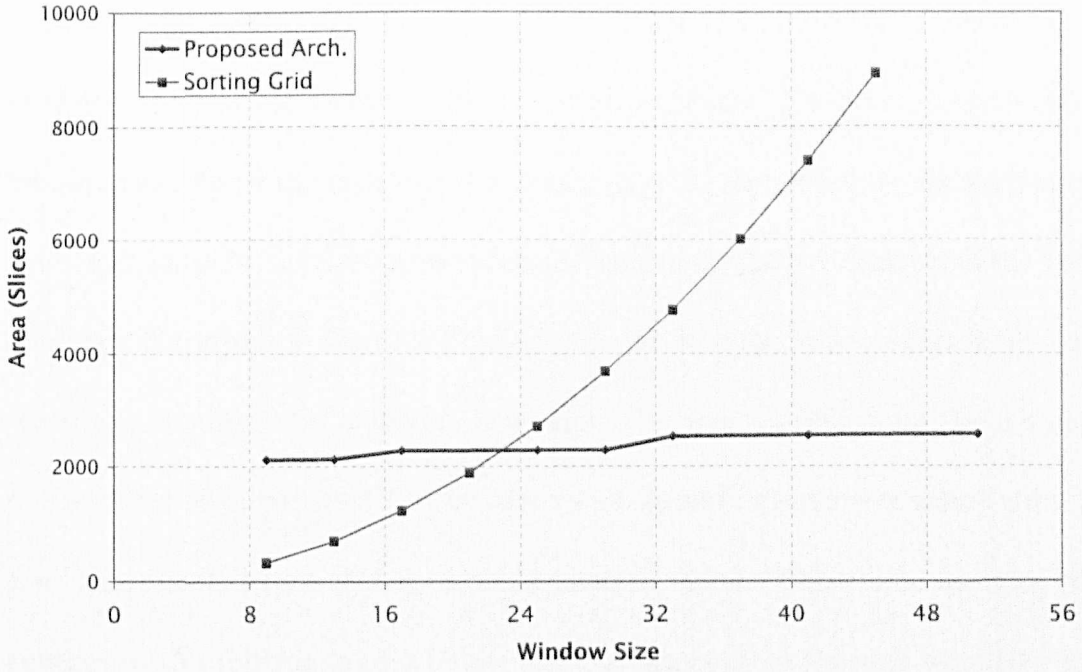


Figure 6.10: Comparison of area requirements for proposed algorithm and sorting grid.

Figure 6.10 shows how this compares very favourably with the area usage of the standard sorting grid architecture that was also implemented. The sorting grid architecture's area requirement increases exponentially with regard to window size. The point at which the proposed architecture becomes more efficient is at a window size of approximately 23 samples. Note that other sorting algorithms can be used. However, the best case is of order $N \log(N)$, so the proposed algorithm remains advantageous, especially for large window sizes.

The next variation was to implement generalised comparators. In these implementations, the median index is computed automatically from the value of the counter in the last bin. Recall that the last bin in a cumulative histogram contains the count of the total number of samples in the system. This can be

halved and incremented to give the median index. The strength of this system is that it allows for variable window sizes. Clearly, the area requirements will increase, since the comparators cannot now be simplified by the synthesis tools and must be full l-bit comparators, where l is the wordlength of the bin counter. The graph in Figure 6.11 shows how each of the area metrics increases when this modification is made. A window size was selected from the middle of the range of values used for the first set of results[2] and an equivalent circuit was implemented but with a variable median index. There was no need to synthesise the full range of window sizes, as the only difference would be in the FIFO length. The dotted lines in the graph indicate the requirements for the fixed comparator equivalents. The number of Flip Flops remains almost constant since the FIFO is not affected by this architectural change. The LUT usage, however, increases by between 23% and 26%, while the Slice count increases by between 19% and 22%.

All designs were synthesised to run at 72MHz. All designs used 8 Block RAMs to implement the bin selection lookup. Each on-chip 18Kb Block RAM can be configured in a number of width and depth configurations. The shallowest configuration is 512 ×36-bits. Hence a 256×256 memory would require 8 of these side by side.

The final variation of designs was the weighted median implementations. Recall that each sample in this implementation has an associated weight; this

[2]The window sizes used for each of the different wordlengths were 13, 25, 51, 109, 211, 387 and 739 respectively.

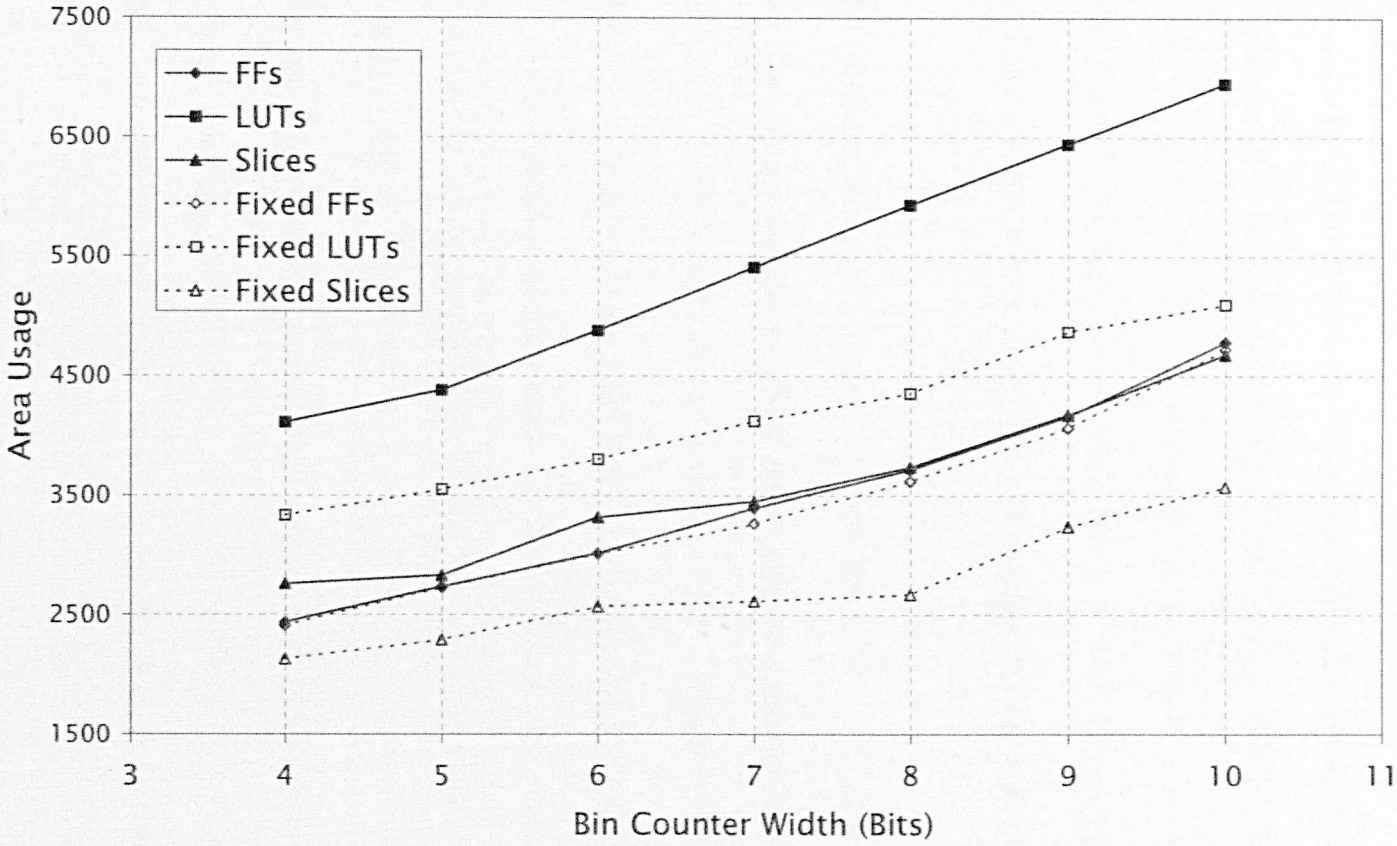


Figure 6.11: Comparison of area requirements for fixed value and variable comparators.

weight is used to update the histogram. This introduces another variable; the width of this weight in bits. Clearly, this will have an effect on the window sizes that can be implemented for each bin size. If the bin width is set to p bits, then for the standard median, it can accommodate a window size of up to $2^p - 1$ samples. For the weighted median, this window size will depend upon the width of the weights. If the weights are given widths of q bits, then the maximum window size for a bin width of p bits is $2^{p-q}-1$. Hence, increasing the width of the weights means wider bins are required for an equivalent window size.

The results of this set of implementations are shown in Figure 6.12. It can be seen that increasing the width of either the bin counter or weight has a similar effect. Furthermore, the area required for a weighted median implementation with weights 2-bits wide is not very different from the generalised version of the standard median filter. As the width of the weights increases the weighted median implementation begins to exceed the generalised median more significantly.

Through developing a parameterised design, it is easy to tailor the implementation to specific requirements in terms of wordlengths and window size. The only assumption that holds for all the above designs is that the input samples are 8-bits wide, as one would find reasonable in the sphere of image and video processing. The extensibility of the original design coupled with full pipelining has meant that all these derivatives could be derived from one architecture, and all can run at 72MHz, returning one result in every clock

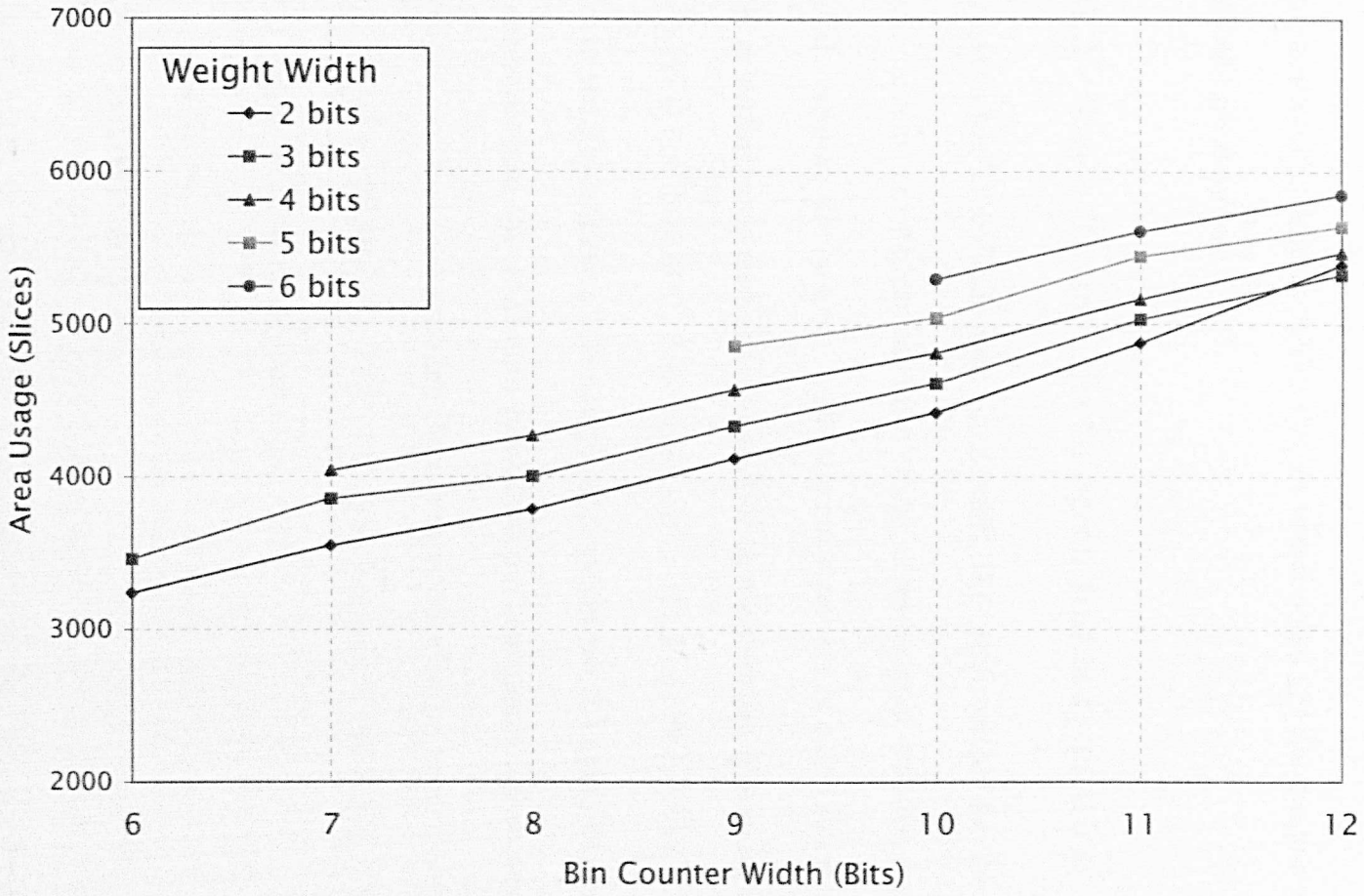


Figure 6.12: Area requirements for various weight and bin width combinations.

cycle. The throughput is thus 72M samples per second. Note that this cannot be converted to frames per second since the implementation is not designed as a spatial filter; it is an arithmetic unit. However, for illustration's sake, the computation throughput of this circuit would equal 234 frames per second for 640×480 pixel images.

6.5.3 Trace Transform Specific Implementation

For use in the Trace transform functionals, the requirements for the weighted median circuit are slightly modified. First, a reset signal is added such that all the bin counters can be reset at the start of a new set of data (a new line). The reset is implemented such that the sample that enters the system in the following cycle is not discarded, since there are no spare cycles between subsequent rows in the Trace image. The sample and weight widths are both set to 8 bits to accommodate the maximum possible input parameters. Furthermore, the circuit is modified to completely ignore samples with an accompanying mask value set to 0. Finally, the FIFO for input samples is discarded, since this is not a sliding window implementation.

6.6 Summary

In this chapter, an alternative implementation of median filtering for arbitrarily large one-dimensional windows was presented. The area required to implement this architecture is of the form $K + \log_2 N$, where N is the window size, thus

it is highly scalable. The design also allows for a flexible window-size that can change from one window to the next. Use of heterogeneous FPGA resources allow the circuitry to be simplified and fully pipelined. The area requirements were compared to that of a standard sorting-grid architecture and show the efficiency of this method for larger windows. For a standard architecture, the area requirements increase exponentially with window size. An extension to weighted median calculation was also shown, that has modest impact on area requirements. A full analysis of area requirements for both fixed-size windows, flexible windows and the weighted median implementation was shown. The presented method is elegant in its flexibility with regards to window size. Of course for very small windows, other techniques may be more compact. However for large windows, or systems where flexibility in window size is needed, or for weighted median calculation, the proposed method is scalable, offers a throughput of 72M Samples/s and uses 15% of the area of the target FPGA.

Given that the weighted-median function is used in a number of Trace transform functionals, this contribution significantly assists in enabling a hardware implementation of the Trace transform. The challenge answered in this chapter was the design of a circuit that can return median and weighted medians for windows of arbitrary length, not necessarily known in advance, in a single cycle. This has been successfully achieved by exploiting the heterogeneous resources on the FPGA. Previous techniques have all assumed fixed window sizes and most architectural optimisations could not be applied to weighted median calculation. This architecture is highly flexible and addresses both.

Chapter 7

Hardware Acceleration of Pseudo 2-Dimensional Hidden Markov Model Decoding

7.1 Introduction

In Chapters 4 and 5, a flexible real-time implementation of the Trace transform was presented. The Trace transform has shown to be applicable to a wide range of application domains as shown in Chapter 3. One such domain is that of face authentication [SPKK05] as touched upon throughout this thesis. It was also noted when discussing applications, in Chapter 3, that the Trace transform is best suited when applied to images free from background clutter or noise. As such it is often applied after a segmentation routine, resulting in a mask as used in the implementation presented thus far.

In this chapter, a Hidden Markov Model decoding implementation will be presented. Such a system can be used to extract frames of interest from a video sequence where a person is present. The resultant frame could then be processed using the Trace transform.

The most successful application of the Hidden Markov Model (HMM) has been in speech recognition, with research going back nearly 20 years [RJ86]. In the computer vision domain, much activity has been seen recently, with the HMM being used for character recognition in deformed text [KA94], template matching [BM04] and face recognition [Nef99]. The strength of the Hidden Markov Model (HMM) is in its ability to cope with deformity to the image [Nef99]. One application of HMM decoding is in person-detection and tracking, when combined with a Kalman filter, as presented in [RWM99, REM00, BR01, BR03].

Unfortunately one of the main difficulties with the use of the Hidden Markov Model is its computational complexity. Implementation in hardware seems an ideal solution to this problem, in order to enable faster processing. While some work has been done on hardware implementation of the Hidden Markov Model for speech recognition [VFJ01], this is the first work to explore a hardware architecture and implementation of the Hidden Markov Model specifically for vision systems.

The work in this chapter deals with accelerating the HMM state decoding, which is the operation used during recognition tasks. This forms part of the algorithm presented in [BR03] for person-tracking. Achieving real-time per-

formance would mean that the algorithm could be applied to a video sequence to extract frames in which a person is present.

7.2 The Hidden Markov Model

The Hidden Markov Model (HMM) is essentially an extension of a standard Markov-process state machine [RJ86]. The idea is that there exists a process which transitions through a number of states. These states are not directly observable, but some other observation can be made that is statistically linked to the state of the process. By knowing the sequence of observations and the properties of the process, the underlying (hidden) state sequence can be deduced.

This is called the "state decoding" problem of HMMs. The information available is as follows:

- $A = \{a_{ij}\}$ where $a_{ij} = Pr(q_j \text{ at } t | q_i \text{ at } t-1)$, the state-transition probabilities
- $B = \{b_j(O)\}$ where $b_j(O) = Pr(O \text{ at } t | q_j \text{ at } t)$, the observation probabilities
- and $\pi = \{\pi_i\}$

where π_i are the initial state probabilities and q_i are the states. [RJ86]

Some important notes for HMMs are that there is only one observation and state-transition per timestep, and the state-transition and observation

probabilities do not change over time.

Recognition using HMMs relies on the Viterbi algorithm [For73] to extract the state sequence from a series of observations. The Viterbi algorithm has been widely researched and efficient implementations in the field of block-convolution decoding and speech-recognition have been proposed [BYC01, ZB03].

The state-decoding problem is that of trying to deduce the transition sequence of hidden states given the sequence of observations. This is done by solving the recursive equations in 7.1 and 7.2. $\delta_t(j)$ computes the probability of being in state j in timestep t, while $\psi_t(j)$ gives the most likely predecessor of state j at timestep t.

$$\delta_t(j) = \max_{0 \leq i \leq N} [\delta_{t-1}(i) \cdot a_{ij}] \cdot b_j(O_t) \tag{7.1}$$

$$\psi_t(j) = \arg\max_{0 \leq i \leq N} [\delta_{t-1}(i) \cdot a_{ij}] \tag{7.2}$$

The state sequence is obtained when δ and ψ are computed for the last timestep. The state with the greatest value of δ is taken to be the final state. The value of ψ for that state is then used to find the predecessor and the backtracking process continues recursively until a full state sequence has been obtained.

It is important to note that the HMM as used in these systems uses offline learning. That is, the model is taught using training data, until accurate parameters are obtained. These parameters are then used in recognition systems

such as ours. The recognition system does not adapt to its input over time, although the reconfigurable nature of FPGAs means that system parameters can be changed by writing a completely new design to the chip.

7.2.1 2-Dimensional Representation

The state sequences discussed so far have all been one dimensional with transitions occurring in the time dimension. However, for application of this theory to images and visual data, the HMM must be extended to two dimensions. This would allow parts of the image to be assigned to different states. A fully connected model, where each state can transition to any other state, is not scalable since the number of connections increases quadratically with the number of nodes. This increases the complexity of the training and decoding nodes quadratically over the one-dimensional approach. Another method is to let the states in a one-dimensional HMM themselves contain HMMs. This is called the embedded Hidden Markov Model [KA94]. The structure can be simplified further by flattening which gives a state-representation as shown in Figure 7.1.

It is important to note that this is not a true 2-dimensional representation, since transitions from column-to-column are not possible. This is called the Pseudo 2-Dimensional Hidden Markov Model [KA94]. This state representation is the one used in the proposed system [BR03], and the efficiency savings gained from this will be shown.

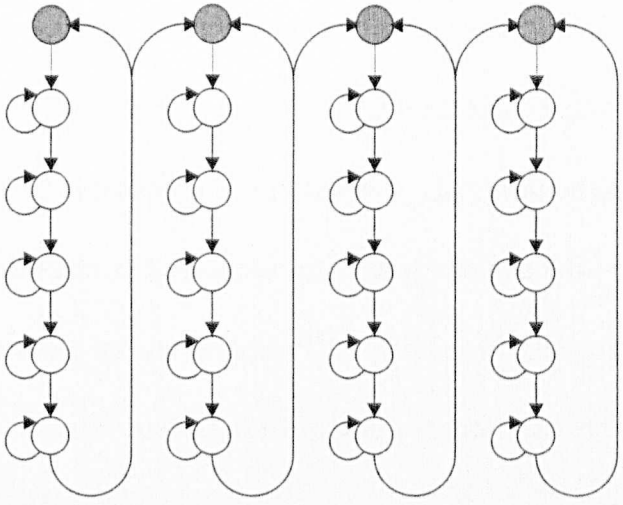


Figure 7.1: State representation of the pseudo 2-dimensional HMM.

7.2.2 System Overview

It is important to consider where the HMM decoding unit fits in to the proposed vision system. As proposed by Rigoll et al [RWM99, REM00, BR01, BR03], the HMM is used to identify the presence of a person. First comes the person-detection phase: through background subtraction, a moving object is extracted. A bounding-box is formed by adding a margin on each side of the moving object. This image segment is then processed with a block based on the Discrete Cosine Transform, using an overlapping sliding window, to extract features. These are the data presented to the pre-trained HMM block as observations, and the block decides whether or not a person is present in the bounding box, by taking into account the number of person states in the extracted sequence.

Once the presence of a person has been established, the system enters the person tracking phase. Segmentation is performed based on the states; the Centre Of Gravity (COG) of the segment is then passed to a Kalman Filter

that predicts the position in the next frame. A new bounding box is formed around the predicted COG and passed to the HMM block to check the presence of a person again. If the person is still present, the parameters of the bounding box are again passed to the Kalman filter to make the next prediction and so on. If the person is no longer present, the system switches back to the person-detection phase. This is summarised in the person-tracking system flowchart shown in Figure 7.2.

Note that while in the person-detection phase, the camera must be stationary for successful segmentation. However, once the system enters the tracking phase, panning and zooming are allowed. This is one of the strengths of this system as compared to many other tracking algorithms. The work in this chapter deals solely with acceleration of the HMM decoding part of the above system.

7.3 Computational Considerations

7.3.1 Log Domain Representation

To decode the state sequence, a multiplication is needed for each predecessor, and one more for multiplying by the observation probability, as seen in (7.1). Given the recursive nature of the equation, dynamic range is an issue that must be considered, since recursively multiplying a number can lead to overflow. One way of overcoming this issue is to perform these calculations in the log-domain. This reduces multiplications to additions and allows the wide dynamic range

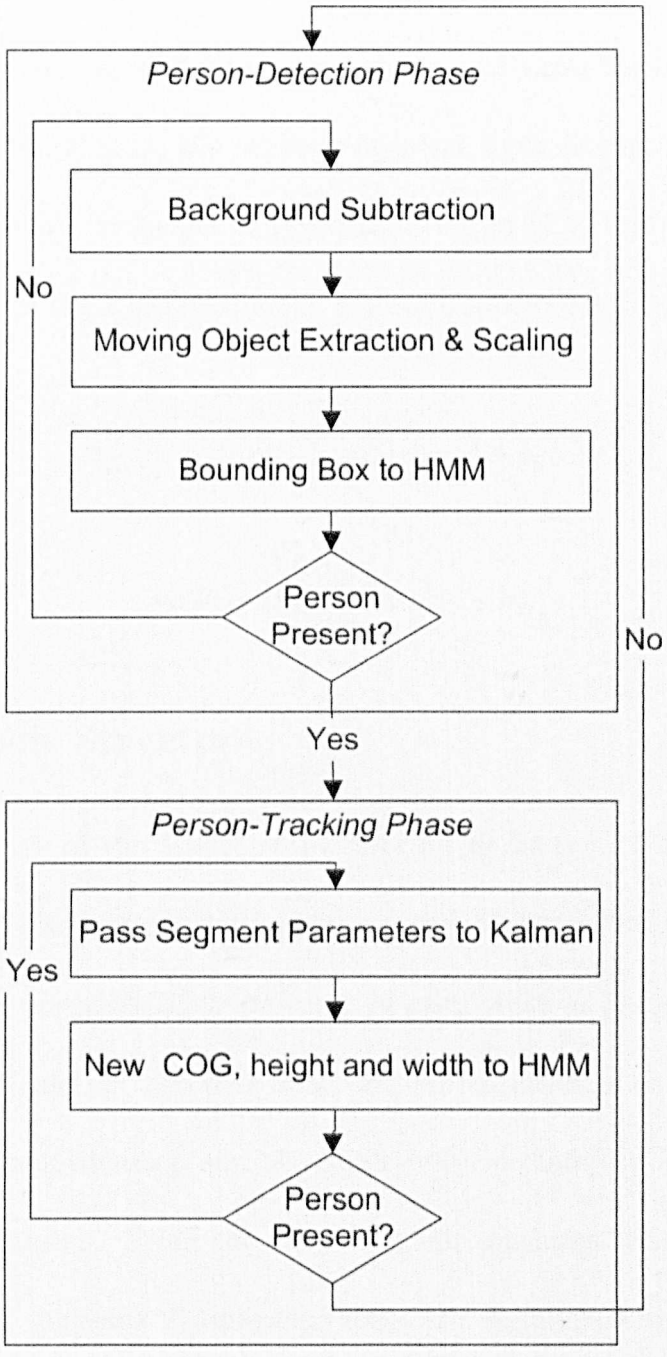


Figure 7.2: The person-tracking system processing flow.

to be represented with fewer bits. Furthermore, since it is relative rather than exact values that decide the state sequence, the relative loss of precision is tolerable and does not impact the results [Mel03].

Given that the logarithm of a probability is always negative, due to the number being less than 1, the result is negated, removing the need for signed arithmetic [Mel03]. Therefore the maximisation in (7.1) and (7.2) becomes a minimisation. In the log domain, the system now needs to compute:

$$\delta_t(j) = \min_{0 \leq i \leq N-1} [\delta_{t-1}(i) + a_{ij}] + b_j(O_t) \tag{7.3}$$

$$\psi_t(j) = \arg\min_{0 \leq i \leq N} [\delta_{t-1}(i) + a_{ij}] \tag{7.4}$$

7.3.2 Trellis Structure

The general form of the Viterbi algorithm for deduction of a state-sequence from a series of observations has been presented. For each timestep, the system must compute the probability of being in each state as defined in (7.1) and (7.2). This calculation depends upon the probabilities of each of the states from the previous timestep and the observation probability for each state in the current timestep. From the equations, for a system with N states, and an observation sequence T timesteps long, the number of multiplications is $(N^2 + 1) \cdot T$. In the proposed system (derived from [BR03]), the number of states is 24 and the typical number of observations per image is in the region of 3000. This gives a total of 1,728,000 multiplications to be completed per

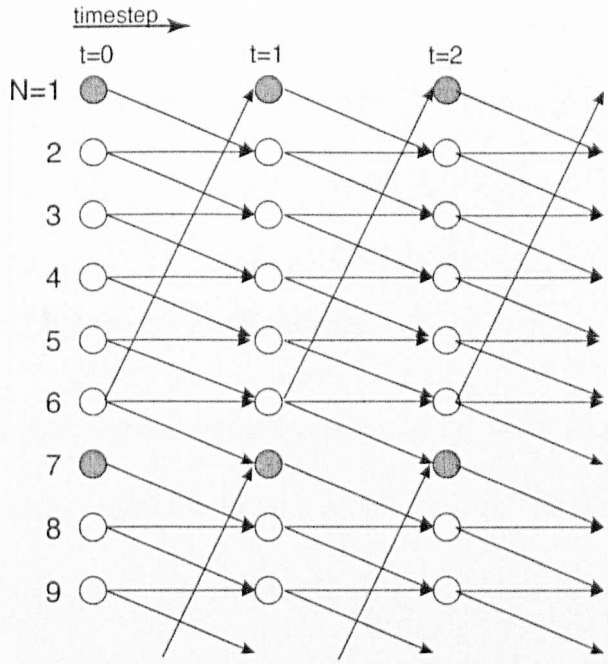


Figure 7.3: Extract from the state-transition trellis for the pseudo 2-dimensional Hidden Markov Model.

frame. For a real-time system running at 25 frames per second, this means over 43 million multiplications per second.

Looking at Figure 7.1, one can see that the state transitions for the pseudo 2D representation are not fully connected. The state transition trellis for this representation is shown in Figure 7.3. Each state only has 2 predecessors. Taking advantage of this would simplify the calculation immensely, reducing the number of multiplications to $(2N+1) \cdot T$. That is a reduction in computation of 90% for these parameters.

One can also deduce that the state transition sequence follows a fixed pattern. The general case is that the 2 possible predecessors for each node N in timestep T are the nodes $N-1$ and N from timestep $T-1$. However, in the case of states 1,7,13 and 19 the predecessors are nodes $N-1$ and $N+5$ from the

previous timestep. Hence, it is possible to design an efficient node-calculation unit that has 2 inputs, one of which depends on which state the result is being computed for.

7.3.3 Algorithmic Parallelism

Another property that suggests hardware would be much more suited to HMM decoding than software is the inherent parallelism in the Trellis. In a normal software implementation, each node within a timestep is calculated in turn, before moving onto the next timestep. This means that the system can only cope with an observation rate that allows it to compute all nodes in the inter-observation time. In this case, 24 calculation-times must complete before the arrival of the next observation.

From the trellis diagram, it can be observed that nodes in one timestep only depend upon values in the previous timestep. This means that more than one node can be calculated in parallel since results from within the same timestep have no effect on each other. In fact, given sufficient resources, all nodes in one timestep could be calculated in parallel. This allows for a higher observation-rate in line with the aim of realtime processing.

7.4 Proposed Architecture

The basic idea of the proposed design is to implement a "decoder node", that goes through each state to compute the δ and ψ values for the current timestep.

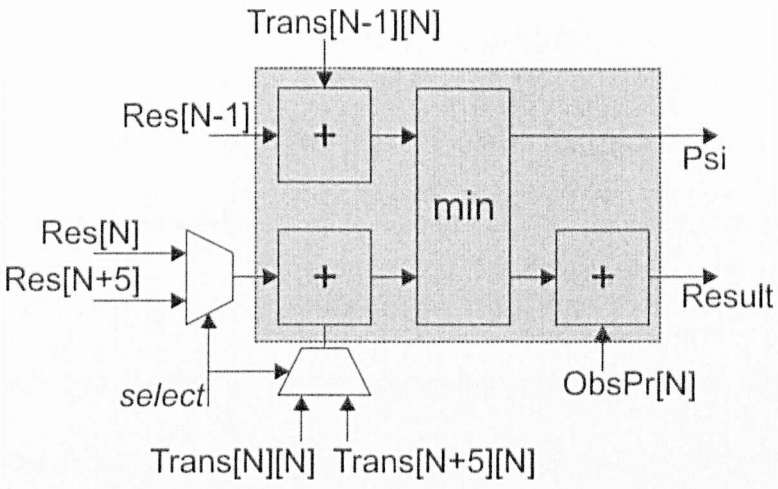


Figure 7.4: The efficient HMM decoder node design.

The node is a simplified solver for (7.3) and (7.4), taking into account the simplifications mentioned in Section 7.3.2. A primitive example of the design is shown in Figure 7.4. The results from the calculations in the previous timestep are fed into the unit. For calculation of the result for state N, the only possible predecessors are states $N-1$, N and $N+5$, as discussed in Section 7.3.2. These are fed as inputs along with the appropriate state transition probabilities and the observation probability for the current state. A *select* signal goes high when the node is computing the results for states 1,7,13 and 19. This causes the values for state $N+5$ from the previous timestep to be used instead of state N. The comparator chooses the minimum of the two values and stores the most-likely predecessor. The observation probability is then added to give the final result for this timestep.

7.4.1 Implementation Considerations

For the purposes of this implementation, only the extraction of the state-sequence from the observation values is considered. As such, the performance of this specific structure for the HMM as compared to others has not been evaluated, nor has the HMM training been considered. Rather, model parameters supplied from some precursory work [Yaq03] on the same system were used.

Since the processing unit is the sole object of concern, the transition probability values have been pre-computed in the log-domain, and the observation probabilities have been assumed to be in the log domain. For a full system implementation, one would have to take into account the area requirements of blocks to convert to and from the log domain. Despite the presence of hard-coded multipliers on the target device, calculations in the log domain were still favoured since this allows problems with dynamic range to be circumvented.

It is essential to understand how the Pseudo-2D HMM maps to an image. In the proposed system, image features are extracted, which form a one-dimensional set of observations. Each observation maps to a position in the image which has been processed using a block based on the DCT, as previously mentioned. Hence, the timestep when looking at the observations is actually a spatial transition in terms of the original image. The word timestep will still be used in this chapter, since this is the preferred terminology when discussing HMMs.

The design was developed and implemented using the Handel-C language and the Celoxica DK compiler. This enabled different levels of parallelism to be tested in a short amount of time with minimal extra effort. The targeted device was a Xilinx Virtex-II 6000 FPGA, on a Celoxica RC300 board, as for the other work in this thesis.

7.4.2 Dataflow considerations

In an implementation as complex as the Viterbi algorithm, organisation of data is paramount to an efficient design. Despite this design being much simpler than a general Viterbi decoder, there were a number of challenges in organising the delivery of data around the system.

The first important data are results from the previous timestep, $\delta_{t-1}(j)$. This is simply an array of 24 values that is copied from the current results, once each time the current timestep completes. The next data item to consider is the observation probability, $b_j(O_t)$. This is again an array of 24 values that changes each timestep. The required value is simply referenced by the number of the state currently being computed. The final and more complex type of data is the transition probabilities that are constant throughout. At first these were stored as a 24×24 array, and referenced by the values of N for this timestep and the previous, but this was too complex. Instead a much simpler approach was developed where each processing node has access to an array of tuples that contains the transition probabilities for the two predecessors, ignoring position, with predecessor selection at a higher level. Hence the node

itself is only the shaded region of Figure 7.4.

In implementing the parallelised versions, some other savings could be made. Consider first, that each hardware node only needs access to the transition elements for the states that it will calculate. More importantly, if one of the parallel nodes will not be computing any of states 1,7,13 or 19, then there is a saving since there is no need to select between two alternative input pairs as in the case of those nodes. This explains why the area requirement does not increase in proportion to the number of nodes, as all nodes above 4 are simpler in their circuitry.

Furthermore, as the level of parallelism increases, the control circuitry becomes more simple, so much so, that in the fully-parallel implementation, there is almost no control circuitry whatsoever. This is the reason for the improved clock speed with a higher number of nodes, as will become clear in the results.

7.4.3 Single-node Implementation

For this implementation a single calculation was implemented. In each timestep, control circuitry uses the node to calculate the results for each state, choosing the correct predecessors. The results are then shifted serially into a shift-register. Once all results had been computed for one timestep, the results are copied, in parallel, to the register holding previous results, ready for calculation of results in the next timestep.

This design takes 24 clock cycles to complete the state calculations for each timestep. The fastest clock rate achievable with the circuit is 36MHz.

7.4.4 Multi-node Implementations

Implementations were completed for 4, 8, 12 and 24 nodes in parallel. These designs take 6, 3, 2 and 1 clock cycle(s), respectively, to complete the calculations for one timestep.

In each case, the appropriate number of nodes is instantiated in parallel. Surrounding logic decides which data to pass to which node. Each node only needs access to whichever data it will process; in the case of the transition probabilities only the necessary tuples were attached to each node.

The implementation for 24 nodes is simpler than for fewer nodes. The reason is that in the case of the 24 parallel nodes, each is hard-wired to the appropriate predecessor registers and transition values, and so there is no control circuitry as such. Since five of the transition probabilities were zero in this case, this removes one of the adders from those nodes. The 12-node version only has binary selections since it only runs for two clock-cycles. Hence there is a significant saving on the multiplexing of signals that causes it to be more area efficient that the 8-node implementation.

The implementations raise an interesting fact: that in the case of this design, the control circuitry is a significant part of the area. This is because a 1-bit adder uses the same amount of resources as a 2-way 1-bit select. Since in these designs the predecessor data is multiplexed into each node, this becomes significant. This is why there is a significant drop in area usage in the graphs from 8 to 12 nodes.

Nodes	Slices	Cycle-time	Cycles/Timestep	M Timesteps/s
1	972	27.033ns	24	1.5
4	2083	34.467ns	6	4.8
8	2271	30.570ns	3	10.9
12	1593	22.112ns	2	22.6
24	1425	14.953ns	1	66.9

Table 7.1: Implementation results for different numbers of nodes instantiated.

7.5 Implementation Results

Implementation results are summarised in Table 7.1 and Figures 7.5,7.6 and 7.7. From the graphs, it can be deduced that the 24-node implementation is most desirable. It is both faster than all other designs and smaller than all except the single-node implementation. However of importance too is the number of cycles needed for a complete result. This swings the result even more in favour of the 24-node implementation as seen in the throughput figures in Table 7.1. For reference a full Viterbi decoder in MATLAB, running on a Pentium 4, 2.4GHz machine, with the same data managed only 1000 results per second. A result rate greater than 200,000 per second[1] would be required for a realtime implementation with 30 frames per second video for the given state representation. The performance presented here equates to over 10,000 frames per second given those parameters.

[1]Calculated from data in[Yaq03].

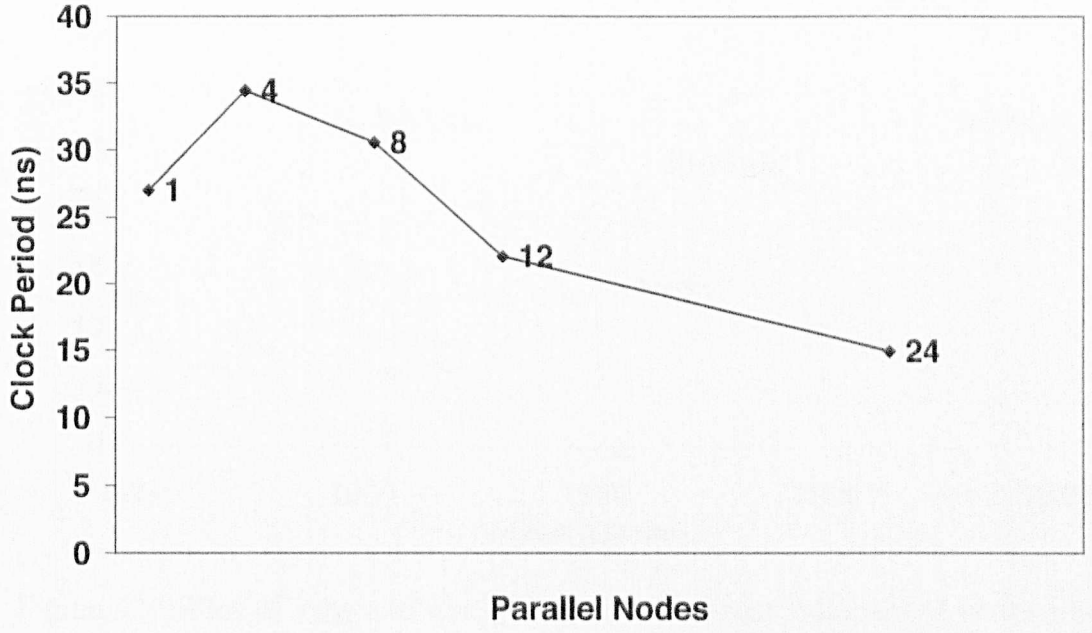


Figure 7.5: Impact of number of nodes on clock period.

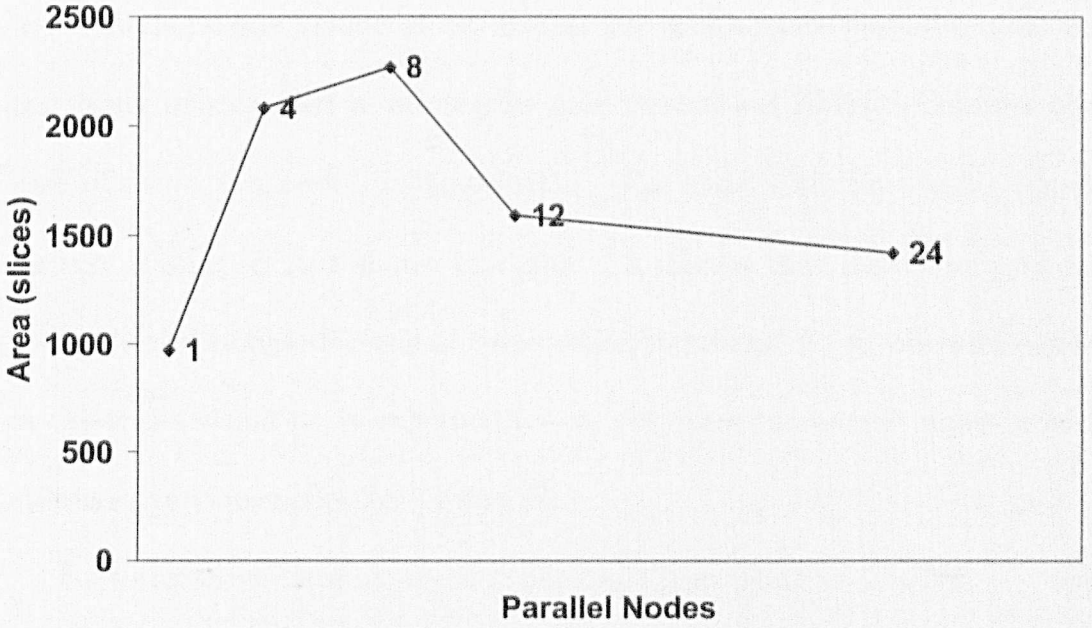


Figure 7.6: Impact of number of nodes on area.

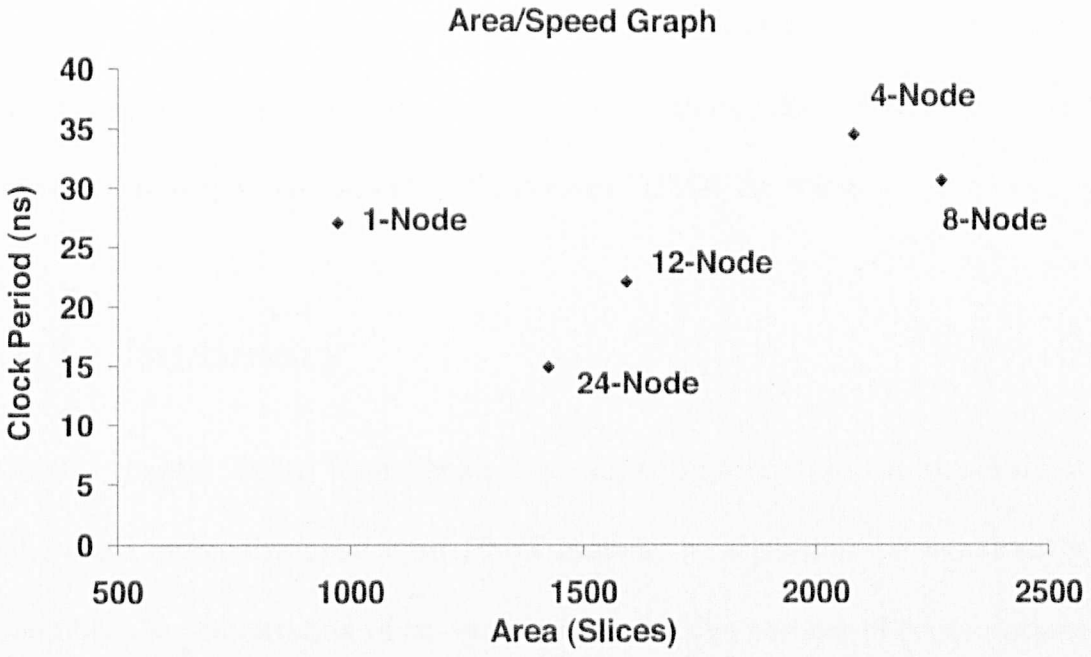


Figure 7.7: Plot of area and clock period for different numbers of nodes implemented.

7.6 Extension to the General Case

The implementation presented here deals with a 24 node HMM, organised as a 4×6 node pseudo 2-dimensional HMM. The architecture presented thus far can easily be extended to arbitrarily sized pseudo 2-d HMMs. Consider the case of an $m \times n$ node implementation. The state transition trellis would be very similar to that shown in Figure 7.3, except that there would be m "special-case" nodes. For one of these nodes, numbered N, its previous nodes in the trellis would be $N-1$ and $N+n$. All other "standard" nodes would still have previous nodes $N-1$ and N.

For different levels of parallelism, one could investigate m, $2m$, $3m$, $\cdots$, nm nodes. However, given the results presented above, it is likely that implementing $m \times n$ nodes would prove most efficient. The decoder node would remain

identical to that presented above, save the change to allow for the larger jump in states for the "special case" nodes. In this manner, the architecture can be used to implement any pseudo 2-dimensional HMM decoding.

7.7 Summary

In this chapter, it has been shown how, taking into account the structure of the state representation for an HMM system, it is possible to significantly simplify the computation of the state sequence. The number of computations is reduced from $(N^2 + 1) \cdot T$ to $(2N + 1) \cdot T$. Different levels of parallelism were also explored, and it was found that increasing the number of nodes not only drastically increases performance, but also has a positive impact on area usage. This is due to the control circuitry becoming simpler as more nodes are implemented in parallel. Such a method could be incorporated into a system to extract frames of interest to be processed by the Trace transform. This architecture is general to any pseudo 2-dimensional HMM state representation, and given the high performance, could be used for a larger number of states while still providing real-time performance.

Chapter 8

Conclusion

8.1 Summary

The Trace transform is a relatively recent algorithm, which has been shown to be highly flexible, offering excellent performance in a number of application domains. An obstacle to its widespread adoption has been its computational complexity as discussed in Chapter 3. In this thesis, the first hardware implementation of this algorithm was presented in Chapter 4. The implementation was designed with both the algorithm and the target architecture in mind, resulting in a highly efficient, extensible architecture. By exploiting algorithmic parallelism, and making the simplification from line extraction to image rotation, significant speedup of over two orders of magnitude over software is achieved. More importantly, the architecture was designed to allow for easy swapping of functionals, with minimal impact on the timing of the overall system. The hardware architecture achieves a significant speedup of 75× over

software for three functionals. Adding more functionals increases this factor significantly. It is clear that to achieve real-time performance with the Trace transform, it is necessary for such a hardware architecture to be used. Since the architecture's performance is immune to the addition of further functionals, it also allows the application designer the freedom to use as many functionals as necessary without a performance cost.

The Trace transform is general in that the functionals used in computations are not pre-defined. This results in the correct selection of appropriate functionals being the deciding factor in creating a successful application. The functional space can be extensive, given the fact that the only requirement is that a functional maps a vector to a single number. In order to facilitate a more thorough investigation of the functional space, a framework for designing flexible functionals was developed, as presented in Chapter 5. This was applied to create three functional blocks that could each implement multiple variations of functionals for a face verification application. In order to facilitate flexibility, the embedded memories on the target FPGA were used to provide re-programmability. A configuration register was introduced into each functional to allow for variable datapaths. The timing impact of this extension of the algorithm was negligible, and for more complex functionals, provides an even greater acceleration factor over software. With the three functional blocks presented, 11 functionals from a previous implementation could be calculated with a significant speedup of over 160× over software.

This framework for designing flexible functionals opens the door to a more

thorough investigation of functional performance for a variety of domains. It is now possible to apply the Trace transform to new domains, researching the efficacy of a wide range of functionals, before deciding on those that most suit the specific task at hand.

In designing the functionals, an efficient, flexible implementation of median and weighed-median filters was presented in Chapter 6. The proposed architecture suits the Trace transform due to its flexibility in terms of window size, and efficiency in computing medians over large windows. A large rank of parallel bins maintains an up-to-date cumulative histogram with each input sample that enters the system. The elegance of the design is apparent in the multiple configurations presented, each requiring only minor changes to the overall architecture. This method of histogram generation has separately been applied to histogram equalisation of images by others.

Finally, an acceleration of Pseudo 2-Dimenstional Hidden Markov Model (HMM) decoding was presented in Chapter 7. By considering the state transition probabilities and investigating varying levels of parallelism, real-time performance was achieved, using a simple replicable processing node. This architecture is extensible to any pseudo 2-dimensional HMM decoding. The HMM decoding block has been shown in previous work to be useful in person-detection, and could be used as a predecessor block the the Trace transform in a vision system, since the Trace transform requires prior segmentation for accurate performance.

Modern heterogeneous FPGAs, with the wide array of embedded elements

that they sport, offer an ideal platform for acceleration of vision algorithms. The large amounts of data, and complex processing, that characterise these systems, can be dealt with efficiently through the development of appropriate architectures. Flexibility can also be afforded by using the embedded elements on an FPGA. The Trace transform has proven an ideal candidate for acceleration, yielding excellent performance improvements. As such, an architecture, such as that detailed in this thesis, can be used as an experimentation platform with which to investigate the use of the algorithm in a range of different applications. This architecture removes the previous limitations of the software-only approach, and opens new areas of vision research using the transform.

The spirit of this thesis has been the importance of considering both the algorithm and target architecture in any hardware investigation. It is unfortunate that some designers simply translate a software implementation into hardware. While the acceleration that can be gained from loop-unrolling is welcome, there are often significant factors to be gained from other methods which may not be apparent in a sheet of pseudo-code. Hence it is important for a successful designer to consider the algorithm with full understanding in order to achieve significant speedup.

The heterogeneous resources on modern FPGAs present the designer with a platform that can be exploited in many ways. Throughout this thesis, the various types of resources have been used to implement different aspects of the various architectures. These resources can often provide the solution to a design problem, such as the use of a ROM to provide the massively parallel

selection control in the median filter architecture. Again, a thorough understanding of the target device and the various ways to exploit these resources is a huge advantage in creating a fast, efficient design.

To conclude, the designer's ability to accelerate vision systems in hardware is significantly aided by modern FPGAs' heterogeneous architectures. The real-time performance required for many applications can only be achieved in hardware, and by exploiting these resources through considered design, this can be achieved.

8.2 Future Work

The work in this thesis could be extended in a number of directions. Some suggestions for future work to follow on from that presented here, will be mentioned in this section.

One possibility is to create a fully accelerated application using the hardware architecture presented in Chapter 4. An application such as face authentication would present a challenge, while providing the opportunity to compare performance to the pre-existent software implementation. Other steps in the face authentication application presented in [SPKK05] could be investigated in hardware, or they could be left to software. In any case, the performance and accuracy of the software and hardware systems could be compared.

A graphical interface, allowing the designer to implement flexible functionals could be developed without much difficulty. The interface would present

a simple datapath, where the designer could add lookups and selectable datapaths. The contents of the lookups and configuration registers could then be set, including multiple different configurations to be tried successively.

Adapting the architecture for a more feature-rich platform would offer some more performance improvements. A better transfer interface than USB would remove the current bottleneck in terms of reading the results. Faster memories that can keep up with the FPGA speed would provide a performance boost. Furthermore, extra external memories, or multi-ported memories could be used to compute more rotations in parallel.

Another possible area of work would be to design further flexible functional blocks using the framework presented in Chapter 5, then employ the system to research the efficacy of a large set of functionals for a given novel application. As yet, there has not been a significant investigation of Trace transform functionals that suit specific applications. This hardware architecture serves as an ideal platform for making this contribution.

One area of work enabled by this architecture is to research the efficacy of multiple computationally simple functionals when compared to the more complex ones converted from the software implementation. Given the freedom to use more functionals due to the lack of a performance cost, it may make more sense to use some simple functionals that suit hardware implementation as opposed to some of the complex ones presented in Chapter 5.

Further functionals based on the idea of a weighted sum, and also small filter kernels such as Haar wavelets would pose an interesting area of research.

Such filters are already widely used in computer vision, and could be incorporated into the Trace transform, using the framework presented in this thesis.

It may be worth investigating the performance of the algorithm when only applied to 180° of rotations. With the Radon and Hough transforms as well as the Trace transform for any functional that does not take into account the pixel position in a line, the resultant parameter domain image is odd-symmetric. Doing away with the extra rotations would double performance, allowing real-time performance for larger images. The effect may well be very minimal even for those functionals that do use the pixel position.

It is worth noting some areas where the transform itself could be extended. Given the acceleration achieved using hardware, it is now possible to apply the Trace transform to a video stream. This presents an opportunity to investigate the temporal properties of the transform, and whether any use can be made of these.

Another extension is to use the transform on a processed image with features extracted using standard methods such as edge-detection. There has also been some suggestion that the transform could be applied to small areas of images in a similar method to a sliding window filter. This would enable some sort of local-feature extraction.

Finally, the histogram generation method presented in Chapter 6 could be applied to other fields where live histograms could be useful. One such application is real-time estimation of probability density functions, which could have wide-ranging applications.

Glossary

ASSP Application-Specific Standard Platform, e.g., DSPs such as Texas Instruments TMS320 Platform. Processors that are tailored to a specific application domain.

Bitstream The data file used to configure an FPGA.

Block RAM The embedded memory components on a Xilinx FPGA.

Computational Complexity A measure of how complex an algorithm is to implement. Typically characterises an algorithm by the number of operations with respect to the values of algorithm parameters.

Datapath The path through which data travels in a circuit, including the wires, computational elements and registers.

DSP Digital Signal Processing or Digital Signal Processor, e.g. Texas Instruments TMS320 Platform.

FIFO First-In-First-Out buffer. A buffer that accepts values and propagates them through with each cycle, with samples emerging at the other end in the order in which they arrived.

FPGA Field-Programmable Gate Array. A device that consists of logic and routing that is configurable at runtime to implement and arbitrary circuit.

Functional A function that maps a vector function to a single value. In the case of the Trace transform, the computation that converts a line to a single value in the parameter domain.

GPP General Purpose Processor, e.g. Intel Pentium 4. A processor with a general purpose datapath.

GPU Graphics Processing Unit, e.g. ATI Radeon Series. A processor with a datapath specifically tailored to graphics processing.

Handel-C An extended version of the ANSI C language with constructs to facilitate use in hardware description.

Heterogeneous Architecture A device which contains a variety of different primitive elements, like Slices, embedded multipliers and embedded memories.

HMM Hidden Markov Model. An extension of a standard Markov process, but where the state transitions are unknown.

LUT Look-up Table. A circuit element that takes multiple inputs and stores the resultant output of a single output logic function.

Median The middle value from a set of samples after they have been ordered in terms of magnitude.

Paralellism Describes portions of an algorithm that iterate a variable over multiple values, with each iteration being independent.

Pipelining The process of inserting registers between computational stages. This allows the clock period to be shorter and for the circuit to thus run faster. Some latency is introduced, but this is negligible for complex systems.

RAM Random-Access Memory. Memory that is read and written to in random order.

Reconfiguration Changing the configuration of an FPGA. Can be during runtime (Runtime reconfiguration).

ROM Read-Only Memory. Memory that has fixed contents and can only be read from.

Slice The basic hardware unit on a Xilinx FPGA. Consists of two LUTs and some other logic. Used as the basic unit for area measurement.

Synthesis The process of converting the hardware description to a set of primitive hardware blocks on the target architecture. Done automatically in software.

Thresholding Turning an image into a binary image by setting all values above the threshold to 1 and all values below to 0.

VHDL VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.

Wordlength The width of a signal or memory location in bits. An n-bit word can accommodate values from 0 to $2^N - 1$ in unsigned arithmetic.

Bibliography

[AA05] A.M. Alsuwailem and S.A. Alshebeili. A new approach for real-time histogram equalization using FPGA. In *Proceedings of 2005 International Symposium on Intelligent Signal Processing and Communication Systems. ISPACS.*, pages 397–400, 2005.

[AC97] F.M. Alzahrani and T. Chen. A real-time edge detector algorithm and VLSI architecture. *Journal of Real-Time Imaging*, 3(5):363–378, 1997.

[Alt01a] Altera Corp. *FLEX 10K Programmable Logic Device Family Data Sheet*, 2001.

[Alt01b] Altera Corp. *FLEX 8000 Programmable Logic Device Family Data Sheet*, 2001.

[Alt05] Altera Corp. *Stratix II Device Family Data Sheet*, 2005.

[Alt07] Altera Corp. *Stratix III Device Handbook.* Altera Corp., 2007.

[AP94] G. Angelopoulos and I. Pitas. A fast implementation of two-dimensional weighted median filters. In *Proceedings of 12th In-*

ternational Conference on Pattern Recognition, 9-13 Oct. 1994, volume 3, pages 140–2, 1994.

[APTE+05] R. Aguilar-Ponce, J. Tessier, C. Emmela, A. Baker, J. Das, J. L. Tecpanecatl-Xihuitl, A. Kumar, and M. Bayoumi. Real-time VLSI architecture for detection of moving object using wronksian determinant. In *Proceedings of 48th Midwest Symposium on Circuits and Systems*, volume 1, pages 875–878, 2005.

[BCB02] K. Benkrid, D. Crookes, and A. Benkrid. Design and implementation of a novel algorithm for general purpose median filtering on FPGAs. In *Proceedings of 2002 IEEE International Symposium on Circuits and Systems*, volume 4, pages 425–8, 2002.

[BJRK+03] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P.Sundararajan. A self-reconfiguring platform. In *Proceedings of Field Programmable Logic and Its Applications*, 2003.

[BM04] M. Bicego and V. Murino. Investigating hidden Markov models' capabilities in 2d shape classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(2):281–286, 2004.

[BM08] A. Bharath and M.Petrou, editors. *Reverse Engineering the Human Vision System*, chapter 11: From Algorithm to Architecture. Artech Publishers, To appear 2008.

[BN97] G.L. Bates and S. Nooshabadi. FPGA implementation of a median filter. In *Proceedings of IEEE TENCON '97 IEEE Region 10 Annual Conference*, volume 2, pages 437–40, 1997.

[BP02] L. Breveglieri and V. Piuri. Digital median flters. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 31(3):191–206, 2002.

[BR01] H. Breit and G. Rigoll. Improved person tracking using a combined pseudo-2D-HMM and Kalman filter approach with automatic background state adaptation. In *Proceedings of International Conference on Image Processing*, volume 2, pages 53–56, 2001.

[BR03] H. Breit and G. Rigoll. A flexible multimodal object tracking system. In *Proceedings of International Conference on Image Processing*, volume 3, pages III–133–6, 2003.

[BT04] A. Burian and J. Takala. VLSI-efficient implementation of full adder-based median filter. In *2004 IEEE International Symposium on Circuits and Systems, 23-26 May 2004*, volume 2, pages 817–20, 2004.

[BY92] M. L. Brady and W. Yong. Fast parallel discrete approximation algorithms for the Radon transform. In *Proceedings of ACM*

Symposium on Parallel Algorithms and Architectures, pages 91–99, 1992.

[BYC01] Jung-Gi Baek, Sang-Hun Yoon, and Jong-Wha Chong. Memory efficient pipelined Viterbi decoder with look-ahead trace back. In *The 8th IEEE International Conference on Electronics, Circuits and Systems, 2001. ICECS 2001.*, volume 2, pages 769–772, 2001.

[CA05] S. Chandrasekaran and A. Amira. High speed /low power architecures for the finite radon transform. In *Proceedings of International Conference on Field Programmble Logicn and Applications (FPL)*, pages 450–455, 2005.

[CCH96] Chun-Te Chen, Liang-Gee Chen, and Jue-Hsuan Hsiao. VLSI implementation of a selective median filter. *IEEE Transactions on Consumer Electronics*, 42(1):33–42, 1996.

[CCL07] B. Cope, P.Y.K. Cheung, and W. Luk. Bridging the gap between FPGAs and multi-processor architectures: A video processing perspective. In *Application-specific Systems, Architectures and Processors (ASAP)*, July 2007.

[Cel] Celoxica Ltd. Celoxica Handel-C and DK Design Suite.

[CH02] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[DDM+05] G. De Ruvo, P. De Ruvo, F. Mariano, G. Mastronardi, P.L. Mazzeo, and E. Stella. A FPGA-based architecture for automatic hexagonal bolts detection in railway maintenance. In *Proceedings of the Seventh International Workshop on Computer Architecture for Machine Perception (CAMP)*, pages 219–224, 2005.

[Dea83] S.R. Deans. *The Radon Transform and Some of its Applications.* John Wiley and Sons, 1983.

[FBCL06] S. A. Fahmy, C.-S. Bouganis, P.Y.K. Cheung, and W. Luk. Efficient realtime FPGA implementation of the Trace transform. In *Proceedings of International Conference on Field Programmable Logic and Its Applications*, 2006.

[FBCL07] S.A. Fahmy, C.-S. Bouganis, P.Y.K. Cheung, and W. Luk. Real-time hardware acceleration of the trace transform. In *Journal of Real-Time Image Processing (Submitted)*, 2007.

[FCL05a] S.A. Fahmy, P.Y.K. Cheung, and W. Luk. Hardware acceleration of hidden markov model decoding for person detection. In

Proceedings of Design, Automation and Test in Europe (DATE), volume 3, pages 8–13, 2005.

[FCL05b] S.A. Fahmy, P.Y.K. Cheung, and W. Luk. Novel FPGA-based implementation of median and weighted median filters for image processing. In *Proceedings of International Conference on Field Programmable Logic and Applications*, 2005.

[For73] G. D. Forney. The Viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[FVS05] M. T. Frederick, N. A. VanderHorn, and A. K. Somani. Real-time h/w implementation of the approximate discrete radon transform. In *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 399–404, 2005.

[Gig04] P. Gigliotti. Implementing barrel shifters using multipliers. Application Note XAPP195, Xilinx Inc., 2004.

[GL01] S. Guo and W. Luk. An integrated system for developing regular array designs. *Journal of Systems Architecture*, (47):315–37, 2001.

[HCWC06] P.-Y. Hsiao, C.-H. Chen, H. Wen, and S.-J. Chen. Real-time realisation of noise-immune gradient-based edge detector. In

IEE Proceedings of Computers and Digital Techniques, volume 153, pages 261–269, 2006.

[HFC95] L. Hayat, M. Fleury, and A.F. Clark. Two-dimensional median filter algorithm for parallel reconfigurable computers. *IEE Proc. Vision, Image and Signal Processing*, 142(6):345–50, 1995.

[HGR93] D.P. Huttenlocher, G.Klanderman, and W. Rucklidge. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–63, 1993.

[HS92] R.M. Haralick and L.G. Shapiro. *Computer and Robot Vision*, volume I. Addison-Wesley Publishing Corporation, 1992.

[HWCC05] P.-Y. Hsiao, H. Wen, Y.-P. Chen, and S.-J. Chen. Real-time implementation of noise-immune gradient-based edge detection. In *Proceedings of the International Symposium on Signals, Circuits and Systems (ISSCS)*, volume 2, pages 633–636, 2005.

[IB98] M. Isard and A. Blake. CONDENSATION-conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.

[ITNI06] K. Irick, T. Theocharides, V. Narayanan, and M. J. Irwin. A real time embedded face detector on FPGA. In *Proceedings of Fourtieth Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pages 917–920, 2006.

[JSC06] J. R. Jen, M. C. Shie, and C. Chen. A circular Hough transform hardware for industrial circle detection applications. In *Proceedings of 1st IEEE Conference on Industrial Electronics and Applications (ICIEA)*, pages 1–6, 2006.

[KA94] S-S Kuo and O.E. Agazzi. Keyword spotting in poorly printed documents using pseudo 2-d hidden Markov models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(8):842–848, 1994.

[KIH+81] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro. Motion-compensated interframe coding for video conferencing. In *IEEE 1981 National Telecommunications Conference*, pages 5–3, 1981.

[KOA90] M. Karaman, L. Onural, and A. Atalar. Design and implementation of a general-purpose median filter unit in CMOS VLSI. *IEEE Journal of Solid-State Circuits*, 25(2):505–13, 1990.

[KP90] T. Komarek and P. Pirsch. VLSI architectures for hierarchical block matching algorithms. In *IEEE International Symposium on Circuits and Systems*, pages 45–8, 1990.

[KP98] A. Kadyrov and M. Petrou. The Trace transform as a tool to invariant feature construction. In *Fourteenth International Con-*

ference on Pattern Recognition, 1998. Proceedings., volume 2, pages 1037–1039, 1998.

[KP01] A. Kadyrov and M. Petrou. The Trace transform and its applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(8):811–828, August 2001.

[KP03] A. Kadyrov and M. Petrou. Object signatures invariant to Affine distortions derived from the Trace transform. *Image and Vision Computing*, 21:1135–1143, 2003.

[KP06] A. Kadyrov and M. Petrou. Affine parameter estimation from the Trace transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(10):1631–1645, October 2006.

[LCTW97] Yeong-Kang Lai, Liang-Gee Chen, Tsung-Han Tsai, and Po-Cheng Wu. A novel scalable architecture with memory interleaving organization for full search block-matching algorithm. In *Proceedings of 1997 IEEE International Symposium on Circuits and Systems*, volume 2, pages 1229–32, 1997.

[LW96] Chun-Hung Lin and Ja-Ling Wu. Genetic block matching algorithm for video coding. In *Proceedings of the Third IEEE International Conference on Multimedia Computing and Systems, 17-23 June 1996*, pages 544–7, 1996.

[LZL94] Reoxiang Li, Bing Zeng, and M.L. Liou. A new three-step search algorithm for block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4):438–42, 1994.

[LZP03] Peihua Li, Tianwen Zhang, and A.E.C. Pece. Visual contour tracking based on particle filters. *Image and Vision Computing*, 21(1):111–23, 2003.

[Mat] Mathworks Inc. MATLAB and Simulink for Technical Computing. software.

[MB04] A. Mitra and S. Banerjee. A regular algorithm for real time Radon inverse Radon transform. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 105–108, 2004.

[MCC07] G.W. Morris, G.A. Constantinides, and P.Y.K. Cheung. ROM to DSP block transfer for resource constrained synthesis. *IET Proceedings on Computers and Digital Techniques*, 1:17–26, January 2007.

[Mel03] S.J. Melnikoff. *Speech Recognition in Programmable Logic*. PhD thesis, The University of Birmingham, 2003.

[MKS+03] K. Messer, J. Kittler, M. Sadeghi, S. Marcel, C. Marcel, S. Bengio, F. Cardinaux, C. Sanderson, J. Czyz, L. Vandendorpe, S. Srisuk, M. Petrou, W. Kurutach, A. Kadyrov, R. Paredes,

B. Kepenekci, F.B. Tek, G.B. Akar, F. Deravi, and N. Mavity. Face verification competition on the XM2VTS database. In J. Kittler and M.S. Nixon, editors, *Audio- and Video-Based Biometric Person Authentication (AVBPA)*, volume 2688/2003, page 1056, 2003.

[Mod] ModelTech Inc. Modelsim. software.

[MXS07] Mirmehdi, Xie, and Suri, editors. *The Handbook of Texture Analysis*, chapter 11: A tutorial on the practical implementation of the Trace transform. World Scientific Press, 2007.

[Nef99] A. Nefian. *A Hidden Markov Model-Based Approach for Face Detection and Recognition.* Ph.d., Georgia Institute of Technology, 1999.

[NHAS06] D. Nguyen, D. Halupka, P. Aarabi, and A. Sheikholeslami. Real-time face detection and lip feature extraction using field-programmable gate arrays. *IEEE Transactions on Systems, Man and Cybernetics–Part B: Cybernetics*, 36(4):902–912, August 2006.

[Ooi06] M. P.-L. Ooi. Hardware implementation for face detection on Xilinx Virtex-II FPGA using the reversible component transformation colour space. In *Proceedings of the Third IEEE Inter-*

national Workshop on Electronic Design, Test and Applications (DELTA), 2006.

[PB03] S. Paschalakis and M. Bober. A low cost FPGA system for high speed face detection and tracking. In *Proceedings of International Conference on Field Programmable Technology (FPT)*, pages 214–221, 2003.

[PK04] M. Petrou and A. Kadyrov. Affine invariant features from the Trace transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(1):30–44, January 2004.

[PM96] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):313–17, 1996.

[PPAC06] A. Price, J. Pyke, D. Ashiri, and T. Cornall. Real time object detection for an unmanned aerial vehicle using an FPGA based vision system. In *Proceedings of the 2006 IEEE International Conference on Robotics and Automation*, pages 2854–2859, 2006.

[RAE04] T. Ramdas, L.-M. Ang, and G. Egan. FPGA implementation of an integer MIPS processor in Handel-C and its application to human face detection. In *Proceedings of IEEE Region 10 Conference (TENCON)*, volume 1, pages 36–39, 2004.

[RBK98] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):39–51, 1998.

[REM00] G. Rigoll, S. Eickeler, and S. Muller. Person tracking in real-world scenarios using statistical methods. In *Fourth IEEE International Conference on Automatic Face and Gesture Recognition, 2000. Proceedings.*, pages 342–347, 2000.

[Ric90] D.S. Richards. VLSI median filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(1):145–53, 1990.

[RJ86] L. Rabiner and B. Juang. An introduction to Hidden Markov Models. *IEEE ASSP Magazine*, 3(1):4–16, 1986.

[Rus02] John C. Russ. *The Image Processing Handbook, Fourth Edition.* CRC Press, 2002.

[RV04] D. V. Rao and M. Venkatesan. An efficient reconfigurable architecture and implementation of edge detection algorithm using Handle-C (sic.). In *Proceedings of the International Conference on Information Techntology: Coding and Computing (ITCC)*, volume 2, pages 843–847, 2004.

[RWM99] G. Rigoll, B. Winterstein, and S. Muller. Robust person tracking in real scenarios with non-stationary background using a statis-

tical computer vision approach. In *Second IEEE Workshop on Visual Surveillance, 1999.*, pages 41–47, 1999.

[Sai04] M. Saini. Unleash your creativity with embedded Linux on Virtex-II Pro FPGAs. *Xilinx Xcell Journal*, (48):38–42, 2004.

[SCHA92] E. Shieh, K. W. Current, P. J. Hurst, and I. Agi. High-speed computation of the radon transform and backprojection using an expandable multiprocessor architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(4):347–360, 1992.

[Sed06] N. P. Sedcole. *Reconfugrable Platform-Based Design in FPGAs for Video Image Processing.* PhD thesis, University of London, 2006.

[SI94] V. A. Shapiro and V. H. Ivanov. Real-time Hough/Radon transform: algorithm and architectures. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, volume 3, pages 630–634, 1994.

[Smi07] A. M. Smith. *Heterogeneous Reconfigurable Architecture Design: An Optimisation Approach.* PhD thesis, Imperial College London, University of London, 2007.

[SN96] G. Strang and T. Nguyen. *Wavelets and Filter Banks.* Wellesley College, 1996.

[Sor85] H. W. Sorenson. *Kalman Filtering: Theory and Application.* IEEE Press, New York., 1985.

[SPKK03] S. Srisuk, M. Petrou, W. Kurutach, and A. Kadyrov. Face authentication using the Trace transform. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 1, pages 305–312, 2003.

[SPKK05] S. Srisuk, M. Petrou, W. Kurutach, and A. Kadyrov. A face authentication system using the Trace transform. *Pattern Analysis and Applications*, 8(1-2):50–61, 2005.

[STZ05] D. Siškovičová, J. Turán, and Z.Bojkovič. Invariant image recognition using trace transform and function of autocorrelation. In *Proceedings of EUROCON*, pages 187–190, 2005.

[SV07] L. Sterpone and M. Violante. A new FPGA-based edge detection system for the gridding of dna microarray images. In *Proceedings of the Instrumentation and Measurement Technology Conference (ITMC)*, pages 1–6, 2007.

[Syn] Synplicity Inc. Synplify Pro. software.

[TCJ02] Jen-Chieh Tuan, Tian-Sheuan Chang, and Chein-Wei Jen. On the data reuse and memory bandwidth analysis for full-search block-matching VLSI architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(1):61–72, 2002.

[TCW+05] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *IEE Proceedings. Computers and Digital Techniques*, 152(2):193–207, March 2005.

[TMJF06] F. J. Toledo, J. J. Martínez, J.Garrígós, and J. Ferrández. Skin color detection for real time mobile applications. In *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 721–724, 2006.

[Tof96] Peter Toft. *The Radon Transorm: Theory and Implementation.* PhD thesis, Technical University of Denmark, 1996.

[TT07] T. Tuan and S. Trimberger. The power of fpga architectures. *Xilinx Xcell Journal*, (60):12–15, 2007.

[TZFO05] J. Turán, Z.Bojkovič, P. Filo, and L. Ovseník. Invariant image recognition experiment with trace transform. In *TELSIKS 2005*, pages 189–192, 2005.

[VFJ01] F. L. Vargas, R. D. R. Fagundes, and D. B. Junior. A FPGA-based Viterbi algorithm implementation for speech recognition systems. In *Proceeedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 2, pages 1217–1220, 2001.

[VJ01] P. Viola and M. J. Jones. Robust real-time object detection. In *Proceedings of IEEE Workshop on Statistical and Computational Theories of Vision*, 2001.

[VRSPGP02] M.A. Vega-Rodríguez, J.M. Sánchez-Pérez, and J.A. Gómez-Pulido. An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems. In *Proceedings of the 10th Mediterranean Conference on Control and Autmoation*, 2002.

[WBC04] Y. Wei, X. Bing, and C. Chareonsak. FPGA implementation of AdaBoost algorithm for detection of face biometrics. In *Proceedings of IEEE International Workshop on Biomedical Circuits and Systems (BioCAS)*, pages S1.6– 17–20, 2004.

[WS03] A. Wu and S. So. VLSI implementation of genetic four-step search for block matching algorithm. *IEEE Transactions on Consumer Electronics*, 49(4):1474–81, 2003.

[Xil] Xilinx Inc. System Generator for DSP. software.

[Xil99a] Xilinx Inc. *Xilinx XC4000 Datasheet*, 1999.

[Xil99b] Xilinx Inc., San Jose. Virtex 2 platform FPGA handbook, 1999.

[Xil99c] Xilinx Inc., San Jose. Virtex platform FPGA handbook, 1999.

[Xil04] Xilinx Staff. Celebrating 20 years of innovation. *Xilinx Xcell Journal*, (48):14–16, 2004.

[Xil07a] Xilinx Inc. *Virtex-4 User Guide*, 2007.

[Xil07b] Xilinx Inc. *Virtex-5 User Guide*, 2007.

[Yaq03] M.M Yaqoob. *Object Tracking Algorithms and Implementations.* M.Sc. thesis, Imperial College London, 2003.

[YH95] Hangu Yeo and Yu Hen Hu. A novel modular systolic array architecture for full-search block matching motion estimation. In *1995 International Conference on Acoustics, Speech, and Signal Processing, 9-12 May 1995*, volume 5, pages 3303–6, 1995.

[YJS06] A. Yilmaz, O. Javed, and M. Shah. Object tracking: A survey. *ACM Computing Surveys*, 38(4), 2006.

[YLC99] Hyeong-Seok Yu, Joon-Yeop Lee, and Jun-Dong Cho. A fast VLSI implementation of sorting algorithm for standard median filters. In *Twelfth Annual IEEE International ASIC/SOC Conference, 15-18 Sept. 1999*, pages 387–90, 1999.

[ZB03] Y. Zhu and M. Benaissa. A novel ACS scheme for area-efficient Viterbi decoders. In *Proceedings of the 2003 International Symposium on Circuits and Systems. ISCAS '03.*, volume 2, pages II–264–II–267 vol.2, 2003.