Computer Vision Algorithms on Reconfigurable Logic Arrays

By

Nalini K. Ratha

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science

1996

Professor Anil K. Jain

Abstract

Computer Vision Algorithms on Reconfigurable Logic Arrays

By

Nalini K. Ratha

Computer vision algorithms are natural candidates for high performance computing due to their inherent parallelism and intense computational demands. For example, a simple 3×3 convolution on a 512×512 gray scale image at 30 frames per second requires 67.5 million multiplications and 60 million additions to be performed in one second. Computer vision tasks can be classified into three categories based on their computational complexity and communication complexity: low-level, intermediate-level and high-level. Special-purpose hardware provides better performance compared to a general-purpose hardware for all the three levels of vision tasks. With recent advances in very large scale integration (VLSI) technology, an application specific integrated circuit (ASIC) can provide the best performance in terms of total execution time. However, long design cycle time, high development cost and inflexibility of a dedicated hardware deter design of ASICs. In contrast, field programmable gate arrays (FPGAs) support lower design verification time and easier design adaptability at a lower cost. Hence, FPGAs with an array of reconfigurable logic blocks can be very useful compute elements. FPGA-based custom computing machines are playing a major role in realizing high performance application accelerators. Three computer vision algorithms have been investigated for mapping onto custom computing machines: (i) template matching (convolution) – a low level vision operation (ii) texture-based segmentation – an intermediate-level operation, and (iii) point pattern matching – a high level vision algorithm. The advantages demonstrated through these implementations are as follows. First, custom computing machines are suitable for all the three levels of computer vision algorithms. Second, custom computing machines can map all stages of a vision system easily. This is unlike typical hardware platforms where a separate subsystem is dedicated to a specific step of the vision algorithm. Third, custom computing approach can run a vision application at a high speed, often very close to the speed of special-purpose hardware. The performance of these algorithms on Splash 2 – a Xilinx 4010 field programmable gate array-based custom computing machine – is near ASIC level of speed. A taxonomy involving custom computing platforms, special purpose vision systems, general purpose processors and special purpose ASICs has been constructed using several comparative features characterizing these systems and standard hierarchical clustering algorithms. The taxonomy provides an easy way of understanding the features of custom computing machines.

To my parents — Thank you for your constant encouragement for higher academic pursuits.

Acknowledgments

I take this oppurtunity to express my deep sense of gratitude to my advisor Dr. Anil K. Jain for having having recruited me into the Ph. D. program and assigning me this wonderful thesis topic. His support, encouragement and above all constant monitoring and guidance kept me involved in the project leading to successful completion of my degree. I consider myself very lucky for having worked with such a great personality and mentor.

I wish to thank my committee members Dr. Lionel M. Ni, Dr. John Weng, Dr. Diane T. Rover and Dr. V. Mandrekar for serving on my committee and contributing towards enhancing the quality of my thesis in several ways.

This research work was supported by financial assistance from the Department of Defense. I acknowledge their support. In particular, help from Dr. D. A. Buell, Dr. Jeff Arnold and Mr. Brian Schott of Center for Computing Research, Bowie, Maryland, is greatly appreciated. Several long hours of discussion with Brian and his personal help in making me understand programming on Splash 2 has helped me in a big way. In addition, I wish to thank Dr. Sea H. Choi for his assistance in VHDL related problems. As a project investigator, Dr. Rover was very helpful, easily accessible and took care of many issues to make the project very successful. I thank her for her assistance.

The Pattern Recognition and Image Processing (PRIP) laboratory provided one of the finest and up-to-date computing facilities for this reseach. I thank the PRIP Lab managers Lisa Lees, Hans Dulimarta and Karissa Miller for their dedicated efforts in making the PRIP lab a great place to work.

I have personally benefited from all the PRIPpies, both past and present, for their help and support over my stay at MSU. I thank them all. Special thanks are due to Aditya and Prasoon for their help in proof reading my draft version of the thesis.

Last but not the least, I thank my wife Meena for her unfailing support, understanding and help in successful completion of this thesis. Her encouragement kept me motivated even during the most difficult periods.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of *computer vision* is to automatically construct a description of a given scene from an analysis of the sensed images of the scene. The sensed images can be a single image taken from a single camera, multiple views of the scene (for example, in binocular stereo) using multiple cameras or a sequence of images of the same scene taken over a period of time (as in video sequences or satellite images) using single or multiple cameras. The description of the scene consists of the identity and localization (position and orientation of an object) of the objects present in the scene based on their measured physical attributes (features). In this regard, the goal of image understanding or computer vision differs from that of image processing which involves image-to-image transformations without arriving at a description of the scene. Simply stated, computer vision aims at providing visual capabilities to a machine. It includes techniques from image processing, exploratory data analysis, statistical pattern recognition, cognitive science and artificial intelligence.

Designing robust and general purpose computer vision systems is a challenging task [37, 118]. A number of difficult imaging conditions as well as scene and object complexities are encountered in practice. These non-ideal and confounding conditions arise due to (i) improper lighting, (ii) shadow, (iii) occlusion, (iv) noise in the sensed image, and (v) assumptions made in object representation strategies. A typical computer vision system involves a front-end image acquisition and a preprocessor, followed by a scene interpreter. The back-end deals with interpreting the scene from the extracted features. One of the main problems in computer vision is to automatically determine a salient set of features that is suitable for describing the scene explicitly. In the literature many attempts have been made to design machine vision systems that mimic a human vision system. But, as the human vision system is extremely complex and not fully understood, these human vision-based models and approaches are not very helpful in designing practical machine vision systems.

The input to a machine vision system is not limited to images in the visible band of the spectrum. Often, infra-red and other non-conventional images are fed into a vision system. A fusion of different types of sensing modalities (e.g., in remote sensing) is not uncommon. The task of a computer vision system is to obtain a high-level description from the input pixels. Depending on the task, a sequence of images or a single image in an appropriate wavelength band is used. For example, in a document image analysis system a single scan of the input document is used where as in motion analysis, a sequence of images is used. Computer vision techniques are being used in a number of practical application domains, including document analysis, bio-medical image analysis, robotics, remote sensing, biometry and industrial inspection. The pixel-to-symbol (scene description) mapping is the inverse of approaches taken in computer graphics, where the aim is to generate an image of a scene from a given description. Computer vision problems are difficult to solve, because, quite often, the solution to the desired “inverse problem” is ill-posed [37, 114]. Secondly, the scene interpretation problem may be ill-defined because a real-world scene does not obey the assumptions of the mathematical models used for image representation and matching. More commonly, the general vision problem is computationally intractable. Over the last three decades, many approaches, theories and methodologies have been developed for analyzing problems in computer vision. But, a general purpose computer vision system is still a dream. In spite of these limitations, many successful machine vision systems have been built to handle problems in specific domains.

1.1 Computer vision methodologies

In order to arrive at a symbolic description of a given scene from its sensed image(s), many methods have been described in the literature [114, 194]. The well-known Marr paradigm is based on the “bottom-up” or data-driven approach. In this method, image interpretation is carried out through a number of stages with an increasing abstract representations. The lower-level features are grouped to arrive at the next higher level scene description. The overall system approach is described in Figure 1.1. In this system, several “vision modules” work independently at the lowest level. The responses of these modules are grouped together to form higher level features for the purpose of recognition. The limitations of this approach are described in [114]. In contrast, the other popular paradigm is that the scene description can be achieved using

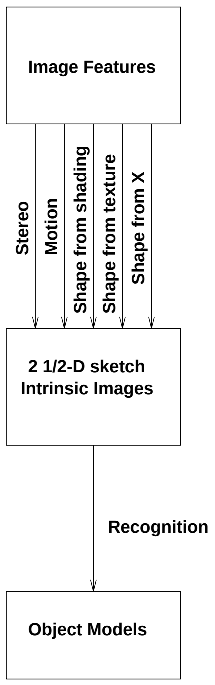


Figure 1.1: Bottom-up approach to designing a vision system. (Adapted from [147])

"top-down" information integration (model-driven). Higher level goals are decomposed into subgoals recursively until the subgoals are reduced to atomic hypothesis verification steps. In reality, a combination of top-down and bottom-up approaches, where top-down constraints are expressed as model-driven predictions that are verified by bottom-up analysis is preferred. This leads to the system design shown in figure 1.2. However, a system design with a feedback path is very difficult to implement. A summary of other methodologies such as active vision and active perception is given in [114].

1.2 Vision task hierarchy

Based on the computational and communication characteristics, computer vision tasks can be divided into a three-level hierarchy, namely, low-level, intermediate-level,

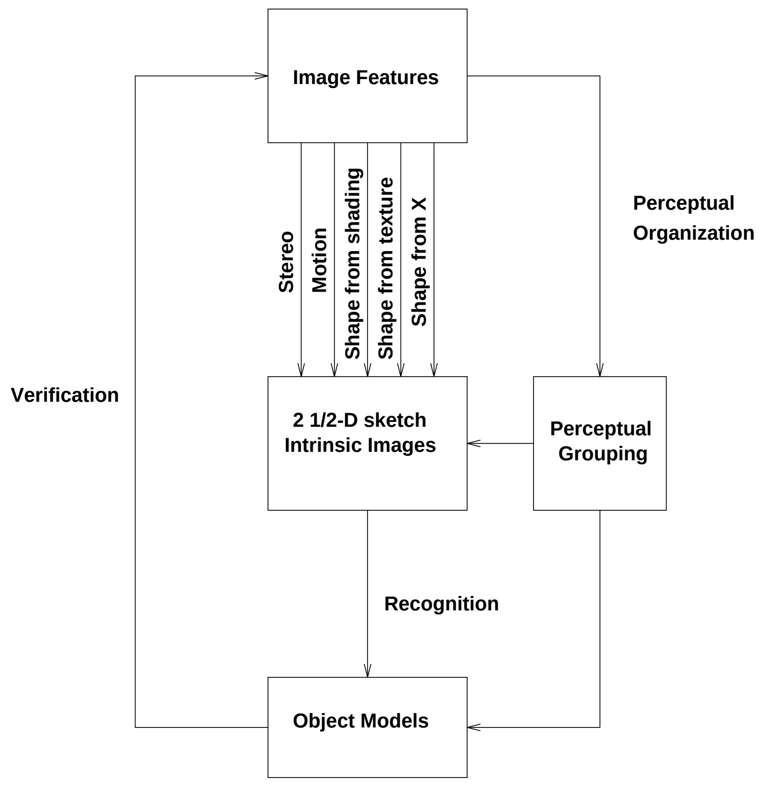


Figure 1.2: Design of a computer vision system with a "feedback" path. (Adapted from [147])

and high-level [225, 47, 226]. Low-level vision tasks consist of pixel-based operations such as filtering, and edge detection. The tasks at this level are characterized by a large amount of data (pixels), small neighborhood operators and relatively simple operations (e.g., multiply and add). The pixel grouping operations such as segmentation, and region labeling are intermediate-level vision tasks. These tasks are again characterized by local data access, but more complex pixel operations. High-level vision tasks are more decision-oriented such as point matching, tree matching and graph matching. These tasks are characterized by non-local data access and non-deterministic and complex algorithms. Several examples of vision tasks belonging to this three-level hierarchy are shown in Table 1.1. The examples under each category

Task level	Computational characteristics	Examples
Low	Small neighborhood data access, simple operations, large amount of data	Edge detection, filtering, image morphology
Intermediate	Small neighborhood, more complex operations	Hough transform, connected component labeling, relaxation
High	Non-local data access, non-deterministic and complex operations	Point matching, tree matching, graph matching, object recognition

Table 1.1: Examples of vision tasks in the three-level hierarchy.

may not be unique. Different researchers may assign the same problem into different categories, e.g., Hough transform is often considered as a low-level task. Similarly, there is some ambiguity about the high-level tasks. The approach taken in this thesis is that if the primary purpose of the task is image enhancement, then it is low-level. The tasks that operate on the pixels to produce symbols (features) are intermediate level tasks. We call the decision making stage as high-level. This classification is purely based on computational and communication criterion described above.

1.3 Computational characteristics of computer vision problems

The visual capabilities endowed to animals and humans look very simple and trivial on the surface, but they turn out to be extremely difficult to describe algorithmically. The computational characteristics of vision algorithms are quite different from other compute intensive problems such as weather forecast models and human genome project as can be seen by the following case studies of problems from each of the three levels of visual tasks.

- Low-level operations:

  Edge detection: Detection of sharp changes in intensity in a gray level input image is an important task. Edges are necessary to describe the raw primal sketch proposed by Marr [151]. Many edge operators are described in the literature starting with simple Robert's edge detector to complex schemes involving regularization-based surface fitting models. A simple edge detector has a computational complexity of $O(N^2M^2)$ for an $N \times N$ image with a $M \times M$ mask. Complex optimization-based techniques, such as simulated annealing-based edge detector [212] have a complexity of $O(2^{N^2})$ for an $N \times N$ image. For a 512×512 image, the total execution time of such algorithms is in the range of several minutes on a SparcStation 20 workstation compared to the desired time of milliseconds (for "real-time" vision applications). A simple 3×3 Sobel edge detector, on the other hand, takes only 0.2 seconds of execution time on a 128×128 image on a SparcStation 20. However, it produces thick edges and for noisy images the performance of Sobel edge detector is poor. The results of GNC-based edge detection, Sobel edge operator and Canny edge operator are shown in Figure 1.3 [79].

  Image compression: Another low-level task is compression and decompression of an image for the purposes of storage and transmission. A commonly adopted method of compression is the JPEG standard. Compressing a 512×512 image

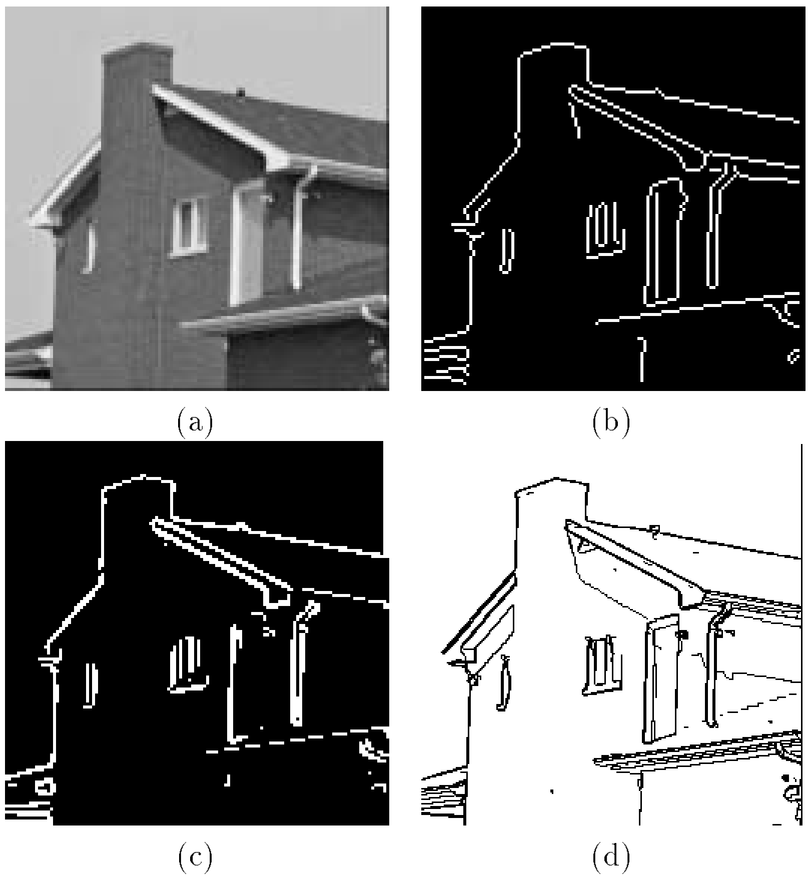
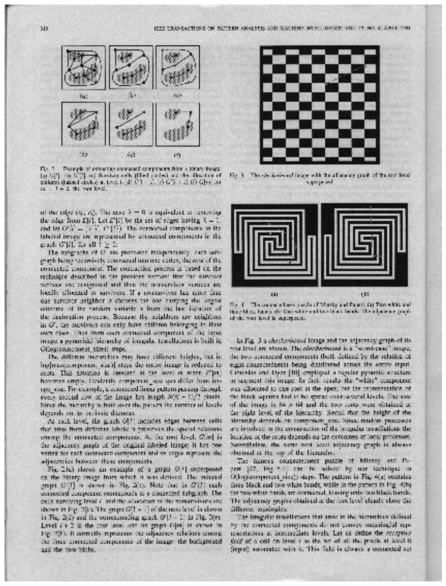


Figure 1.3: Edge detection. (a) Input image; (b) Edge map using GNC; (c) Edge map using Sobel operator (d) Edge map using Canny operator.

using JPEG standard takes several seconds on a SparcStation 20 compared to the desired time of milliseconds.

- Intermediate-level Operations:

  Image segmentation: Obtaining homogeneous regions from an input image helps in obtaining a scene description. Although a general-purpose image segmentation technique still eludes computer vision researchers, many successful domain-specific segmentation techniques are available. A texture-based segmentation method for page layout analysis based on texture discriminating masks has


(a)

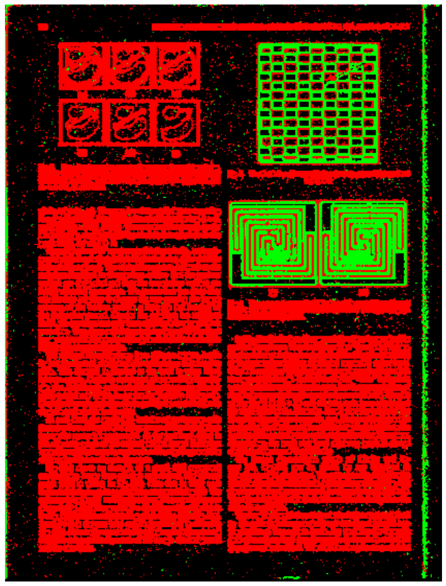

(b)

Figure 1.4: Image segmentation. (a) Input image; (b) Segmented image.

been described in [107]. The algorithm uses a neural network-based approach to learn the convolution masks for segmentation. The learning process involves a gradient descent method of optimization. For a $1,024 \times 1,024$ image, this method takes approximately 250 seconds of execution time on a SPARCstation 20. A three-class image segmentation result is shown in Figure 1.4, where white pixels represent the graphics area, black pixels represent the background and the gray pixels represent the text area of the input image.

Structure and motion estimation: From a stereo image pair, the depth at each image point can be estimated. A multiresolution algorithm has been proposed by Weng et al. in [229]. By constructing a six-layer pyramid from the input image, the disparity information is estimated at every point and projected to a lower level in the pyramid. A non-linear optimization technique is used to provide a more stable solution in the presence of image noise. For a stereo pair of 512×512 images, the depth estimation takes over 10 minutes on a SPARCstation 20. Results of this algorithm are shown in figure 1.5.

- High-level operations:

  At the highest level of visual processing, the tasks of recognition and matching are carried out. Many knowledge-based approaches fall under this category. Typically, the input to this stage are surfaces, lines and points represented in terms of high-level data structures such as graphs, trees, and point vectors. The matching and recognition tasks are expressed in terms of generic graph isomorphism, sub-graph isomorphism, tree matching or point vector matching. Most of these problems fall into the NP-complete class, hence, they are highly computation intensive. For example, consider the case of matching a query fingerprint with the stored images in a large database. Typically, fingerprint databases contain millions of records and a fingerprint image contains on an average, approximately 65 minutiae features. The matching problem can be posed as a subgraph isomorphism problem which is known to be a NP-complete problem. Using a simple model of the minutiae features and simplifying assumptions, the matching problem can be mapped to a point pattern matching problem. For an average of 65 points per fingerprint, a sequential point matching algorithm takes on the order of 3 hours to match a fingerprint against one million images in a database.

In summary, computer vision problems are often ill-posed, intractable and require substantial computational resources. Many simplifying assumptions are made about

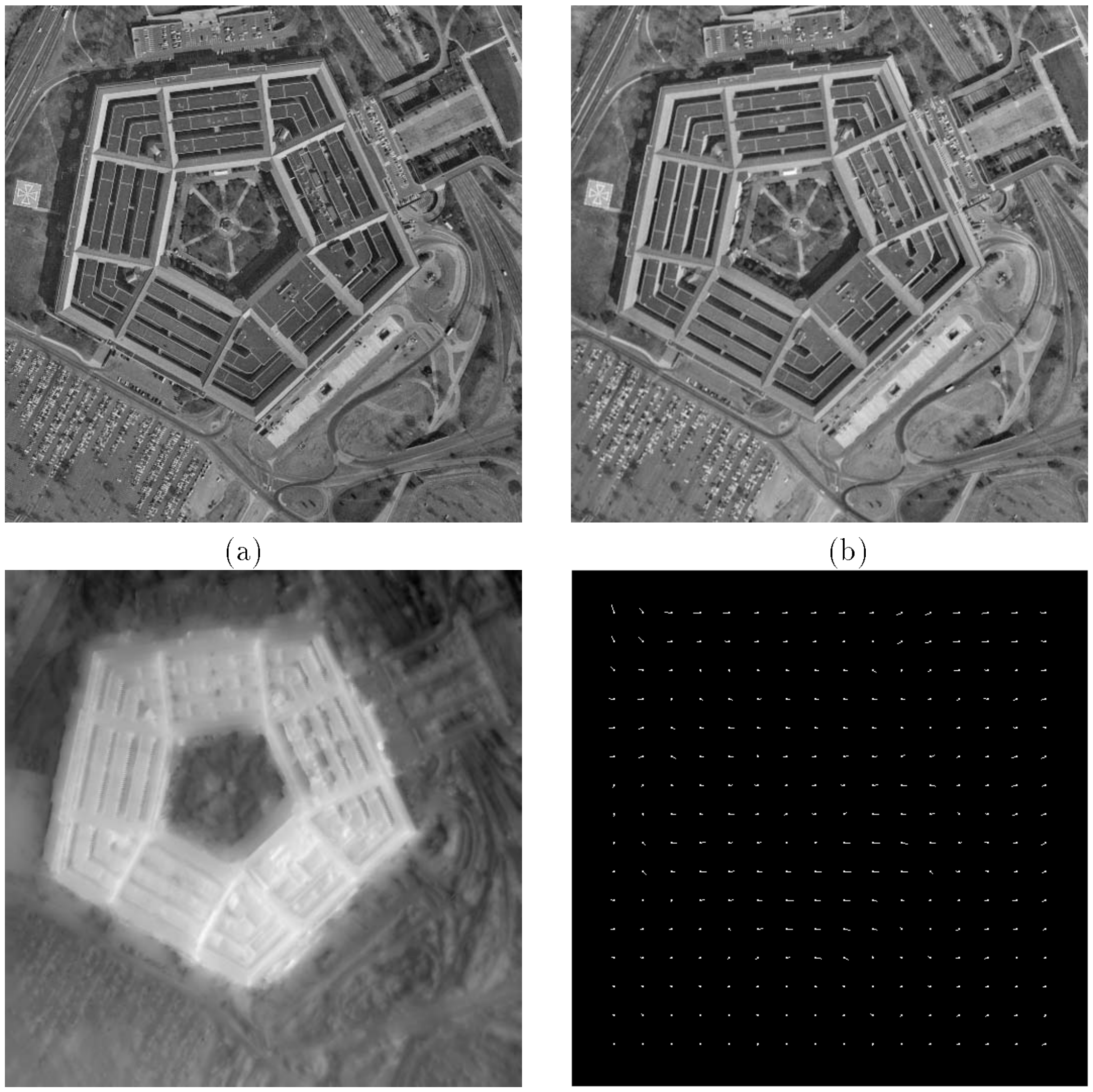


Figure 1.5: Shape from stereo. (a) Left image; (b) Right image; (c) Depth (brighter pixel means closer to the viewer); (d) Displacement vector.

the sensing geometry, light sources, surface geometry and noise sources (e.g., smooth surfaces, Lambertian surfaces and Gaussian noise, etc.). These assumptions make the resulting vision system very brittle in the sense that the system's performance degrades rapidly when the assumptions are violated. In order to overcome these shortcomings, complex processing algorithms involving non-linear optimization techniques are used. However, these complex algorithms demand additional computational resources.

1.4 Need for real-time computer vision systems

A system in which the time instant at which the output is produced, after presentation of the input, is critical, is called a real-time system. Shin and Ramanathan [204] have identified three major components and their interplay that characterize a real-time system. Loosely speaking, the system output must meet a time deadline since the output is related to the input changes. Brown and Terzopoulos [30] define real-time computer vision systems as follows: *Real-time computer vision systems refer to computer analysis of dynamic scenes at rates sufficiently high to effect practical visually-guided control or decision making in everyday situation.* Another definition of real-time system is that the response time of the machine vision system may equal or be faster than the response of the human performing the same task. These definitions lead to an expected processing rate of about 10–30 frames per second. Computer vision systems are employed in many time-critical applications such as silicon wafer inspection. Each wafer needs to be inspected and a decision made before the next wafer arrives. Hence, it is essential that the vision processing be done at the data acquisition rate (video frame rate of 30 frames/second). For applications based on video data, processing at this rate is an essential requirement. In military applications such as target detection, the need for real-time processing is highly critical [66]. Many interesting applications such as automatic car license plate reading demands a real-time processing of the moving-vehicle images. The need for real-time processing is also very important in medical image analysis applications such as vision-guided non-invasive surgery. Similar constraints exist in other applications such as compression/decompression of images in multi-media applications. In order to meet the real-time requirements, a frame of image buffer needs to be processed in roughly 33 milliseconds. For a 512×512 gray level image this amounts to a data rate of roughly 7.5 MHz. The vision algorithms described in the previous section demand a very high execution time on a general-purpose computing platform compared to the desired real-time response. Often, the large disparity between the desired response time and actual response time is reduced by using appropriate hardware accelerators [118].

1.5 Architectures for vision

In order to meet the high computational needs of vision systems, many special-purpose architectures have been proposed in the literature. Machines based on Von Neumann architecture are inadequate to meet the computational requirements of vision algorithms. Hence, special-purpose hardware and parallel processing systems are commonly used to meet the computational requirements of computer vision algorithms. Architectures for vision can be classified on the basis of several variables as shown in figure 1.6.

- The architectures for vision can be classified depending on the type of vision algorithm (e.g., low-level, intermediate-level or high-level).

- Yet another way to classify the architectures is based on the instruction and data streams. The two common classes are Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). Typically, architectures for vision algorithms tend to be of the SIMD class for the lower-level algorithms. Parallelism at lower levels is more obvious compared to high-level algorithms. A taxonomy with several vision system examples from each class is given in [150].

- A third way of classification is based on the type of hardware used: Application-specific processor versus general-purpose hardware.

- Fine grained versus coarse grained: Based on the granularity of parallelism, a special purpose processor can be classified as fine-grained or coarse-grained.

A number of special-purpose architectures for vision are discussed in the next chapter.

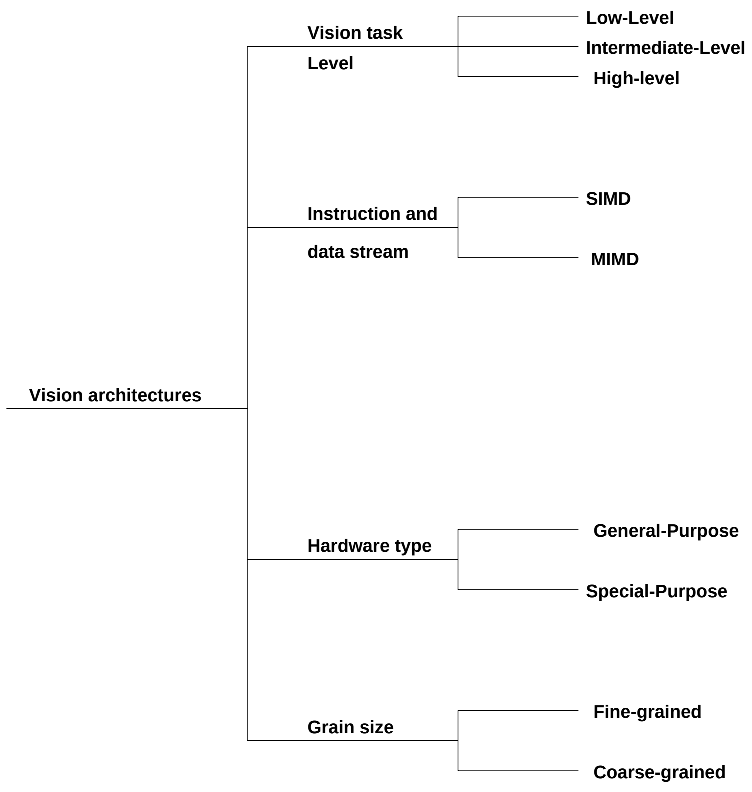


Figure 1.6: Classification of architectures for vision.

1.5.1 Architectural features for vision algorithms

By analyzing several representative problems in computer vision, the following architectural requirements are observed:

- Computational characteristic: For low-level vision algorithms, SIMD and fine-grained architectures are preferred and for high-level algorithms, MIMD and coarse-grained architectures are required.
- Communication: At a lower level, communication is limited to a local neighborhood, but at higher levels the communication tends to be global (non-local).
- High bandwidth I/O: A typical image contains a large amount of data, therefore a high bandwidth I/O is essential to sustain good performance.

- Resource allocation: Speeding up only one stage of the vision system will not result in a significant speedup of the overall performance. Hence, appropriate computational resources should be allocated to all the stages of the algorithm.

- Load balancing and task scheduling: For good performance, the load on different processors should be balanced.

- Fault tolerance: In a multi-processor system, failure of some processing elements should not result in an overall system failure. Therefore, a graceful degradation should be supported.

- Topology and data size independent mappings: Often, a specific processor topology is preferred for an algorithm depending on its communication characteristics. Hence, flexible communication support is essential for mapping many communication patterns. The algorithm mapping should be independent of data size.

These broad characteristics of vision tasks are shown as a pyramid in figure 1.7.

Many novel architectural features have been incorporated in the currently available general-purpose processors to improve their performance. Reduced-instruction set computing (RISC) paradigm is being preferred over complex-instruction set (CISC) computing paradigm. The RISC approach is characterized by a simple instruction set and addressing modes. For performance improvement, pipelining is a very well-known practice. In addition, advanced processors support the following architectural features:

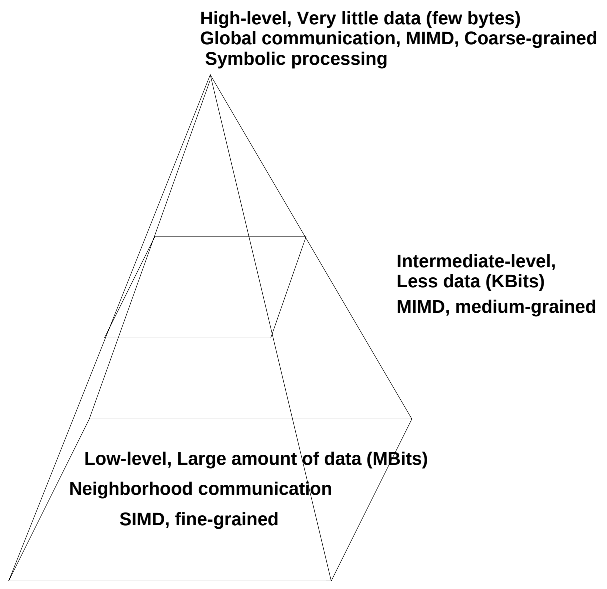

High-level, Very little data (few bytes)
Global communication, MIMD, Coarse-grained
Symbolic processing
Intermediate-level,
Less data (KBits)
MIMD, medium-grained
Low-level, Large amount of data (MBits)
Neighborhood communication
SIMD, fine-grained

Figure 1.7: Vision task pyramid.

- Superscalar: Issue of more than one instruction to more than one execution unit in one clock cycle.
- Superpipeline: An instruction-handling sequence with a large number of stages to allow a faster clock.
- Out-of-order instruction issue and instruction execution.
- Speculative execution.
- Large on-chip instruction and data cache.
- On-chip support for floating point operations and graphics operations.

The main goal of these features is to achieve the optimal level of single clock per instruction (CPI). Many other techniques such as hardware/software branch prediction, and dynamic scheduling are also employed in modern processors. The number of transistors has reached ten million per processor and soon $0.25\mu m$ technology will be available. The clock rates of 500 MHz are already being experimented with and 300 MHz processors are quite common [72]. A comparative evaluation of the four latest RISC processors (Alpha 21164, MIPS 10000, PowerPC 620 and UltraSparc) is given in [72]. These advanced features need hardware support on the chip which can take away a significant amount of silicon space. Tredennick [219] has suggested the following three ways to improve the performance of present day uniprocessors: (i) superscalar, (ii) multiprocessing, and (iii) use of very long instruction word (VLIW).

The various parameters in a processor architecture such as number of stages in a pipeline, size of on-chip cache, and number of functional units in case of superscalar processors are decided by maximizing the average performance over a wide variety of applications. For high performance applications, this design decision is not acceptable as the parameters are not optimized for the application at hand. Experienced programmers often desire to tune instructions to meet their needs in order to satisfy application demands. The undesirable overheads of general-purpose, fixed-instruction set machine can be dispensed with for the sake of performance. Unfortunately, such systems wherein the user can define his own instructions are not yet available. Tredennick [219] proposes a new approach to computing wherein he suggests not to execute low-level operating system routines on the main processor. Instead he suggests a 're-configurable embedded accelerator' that can provide the low-level services for several applications.

1.6 Reconfigurable logic arrays

There are a number of different ways to increase the computational power of processors. Many architectural refinements described earlier, e.g., super-scalar, super-pipelined, high clock rate, etc. [94, 96] demand a high price for retaining the general programmability of these processors.

The user is constrained to think of solving his/her problems within the boundaries of the architecture of the hardware as reflected in the programming language. The language compilers play a major role in exploiting architectural features of the underlying hardware. However, in order to make the programming languages independent of the architecture, the programmers are not exposed to the specific underlying architecture. Even if the architecture is exposed to the user through machine language, the user has never been given the freedom of designing his architecture until recently. The system resources will be effectively used only if the architecture can be tuned to the demands of the application. Until recently, it was not possible to provide architectural specificity coupled with user reprogrammability. Designing an application specific integrated circuit (ASIC) has the advantage of designing a user-specific architecture, but not user reprogrammability of the architecture. The highest speed with a given technology is achievable only if the user can program at the basic gate-level. This is the basis of *reconfigurable arrays.* The reconfigurable arrays are different from reconfigurable architectures which use switching elements to achieve different parallel architectures. Reconfigurable arrays support programmability at the gate level to support architecture customization at the instruction level. There are many types of reconfigurable hardware based on the techniques and technology used to support reconfigurability. Custom computing machines (CCMs) [32] use reconfigurable logic blocks as basic compute elements. Different names have been given to this approach of computing such as FPGA-based computing device, Programmable Active Memory (PAM) [22], and Processor Reconfiguration through Instruction Set Metamorphosis (PRISM) [15].

The main advantage of FPGA-based custom processors is that the logic required for each application can be generated by an appropriate control bit stream. Hence, many diverse applications can use the same hardware. Moreover, no instruction fetch or decode is necessary. Once the control bit stream is loaded, the system is ready to execute the code that is specific to the given application.

Reconfigurable logic arrays can support many of the features demanded by vision algorithms. Evaluating the suitability of reconfigurable logic arrays for vision algorithms is the main objective of this thesis. CCMs are slave processors to a host, so the I/O tasks do not run on them. Often, the time consuming portion of an application program is mapped onto the CCM instead of the whole application. The I/O segments of the application are usually run on the host.

1.7 Hardware-software codesign

The system designer has the dilemma of deciding which portions of an algorithm should be run on a special hardware and which portions should be run on a standard hardware (e.g., a workstation or a microprocessor). Ideally, an application-specific hardware system that could be built without a high cost and without a high turn-around time would be preferred. But, dedicated systems are always coupled with high development time and large investments. At the other end of the spectrum one can use a general-purpose processor at the cost of sacrificing the performance. The designer has to make a cost/performance analysis to arrive at an efficient partition of the problem. This optimization problem is the subject of hardware-software codesign which refers to the process of simultaneously designing both hardware and software to meet some specified performance objectives. In a traditional approach, hardware-software partitions are relatively rigid. In codesign the partition is flexible and can be shifted to meet the changing performance criterion. It is desirable to follow a design cycle that can support modeling complex system designs in hardware and software to arrive at the partition for an overall optimal performance. Recently, Micheli [155] has described a framework for hardware-software codesign. An *embedded system* is a system with a mix of general-purpose processors, dedicated hardware as well as software running on one or more processors in addition to sensors to interact with the environment. Hardware-software codesign is an important method for designing embedded systems. Wolf [230] surveys design of embedded systems which use software running on programmable computers.

1.8 Contribution of the thesis

The main goal of this research is to demonstrate suitability and superiority of custom computing approach for all levels of vision algorithms. For this purpose, the following representative examples of computer vision tasks from low-level, intermediate-level, and high-level vision algorithms have been used. For low-level vision algorithm, a generalized 2-D convolution has been implemented. For intermediate level vision, a texture-based segmentation algorithm is implemented. Point pattern matching is carried out to demonstrate the applicability of custom computing machines for high-level vision algorithms. Special purpose architectures for vision are usually targeted for different levels. The main advantage of using reconfigurable arrays is that the same architecture can be tailored to the demands of a specific level by customizing the instruction set.

Through the design and implementation of these selected algorithms, suitability of custom computing machines for computer vision algorithms has been demonstrated. The other advantage of this framework is the integration of the whole machine vision system on a single platform which is likely to meet the "real-time" requirements. The following advantages and disadvantages have been observed:

- Reconfigurability of the FPGAs can be exploited to meet the requirements of different levels of vision algorithms.
- Multiple styles of parallel programming is possible with multi-FPGA custom computing machines.
- Near ASIC-levels of speed of operation is possible with CCMs.
- The overheads to support reconfigurability comes in the way of a more dense logic.

- High density reconfigurable logic arrays are costlier.
- The CCMs need to be further evaluated for pure floating point-based and non-linear optimization technique-based algorithms.

In addition, a number of features that describe custom computing machines have been studied by way of constructing a taxonomy involving several custom computing machines, general-purpose processors and special-purpose processors.

1.9 Overview of the thesis

The rest of this thesis is organized as follows. Results presented in the area of parallel architectures and parallel algorithms for computer vision are surveyed in Chapter 2. The different design components for reconfigurable architectures are discussed in Chapter 3. The programming flow for such a hardware is quite different from conventional programming style. The programming methods for custom computing machines are also presented in Chapter 3. Splash 2 is one of the earliest custom computing machines. Details of Splash 2 and other recent CCMs are briefly reviewed in Chapter 3. The advantages and disadvantages of these machines compared to standard hardware platforms are also presented. For the purpose of testing our approach, representative algorithms from different stages of computer vision have been chosen. Chapter 4 describes mapping of generalized convolution algorithm as a representative low-level vision task. Image segmentation applied to page layout segmentation based on a two-stage algorithm using convolution and neural network classifier is described in Chapter 5. The feature-based fingerprint matching is presented in Chapter 6. A taxonomy involving CCMs and other well known platforms is given in chapter 7. Finally, the conclusions of this research and directions for future research work in this area are summarized in Chapter 8.

Chapter 2

Parallel Architectures and Algorithms for Computer Vision

A variety of compute-intensive applications (e.g., weather forecasting, computer vision, human genome mapping) have been the main driving force behind parallel processing. A number of these applications are listed as "Grand Challenges" in the Federal High Performance Computing Program [162]. This chapter is devoted to a survey of reported work in the area of parallel architectures and algorithms for computer vision. In the area of custom computing machine for computer vision, only a few results have been reported in the proceedings of the workshops on FPGAs for custom computing machines (FCCM) [98, 99, 100].

The vast amount of work in parallel processing for computer vision will be surveyed under three major sub-areas (i) algorithms; (ii) architectures; and (iii) languages. The general design issues such as theoretical analysis, load balancing and mapping will be briefly reviewed under specific algorithms. Both special-purpose architectures and general-purpose architectures tuned for vision tasks will be covered under the sub-area of architectures. The most well known architecture specific to vision applications is the pyramid architecture. Algorithms and architectures for pyramid-based machines will be reviewed. Many special-purpose VLSI hardwares which have been developed for computer vision will also be described. A large number of commercial image processing accelerators are available in the market. A brief description of the popular ones is included in this chapter. Recent general-purpose parallel processing systems such as SP-2, CM-5 and MasPar-2 are being used for many computer vision tasks. A brief description of the architecture of these two general-purpose parallel architectures is included. Very little work has been reported on parallel languages for computer vision.

2.1 Languages

A fundamental problem in the area of parallel processing is how to express parallelism present in a given algorithm. Many methods are employed to express parallelism explicitly to assist the language compilers. High Performance Fortran (HPF) continues to be the most popular for this task. For image processing related tasks, Brown *et al.* [31] have proposed a language called I-Bol. It treats an image as a tuple of sets. A number of low-level and intermediate-level vision tasks have been implemented using user-defined neighborhood functions. I-Bol is well suited for distributed memory systems and has been implemented over a network of transputer-based processors. The other important parallel language for vision with a particular emphasis to Splash 2 is the dbC [78]. Originally developed for workstation clusters, dbC is now being used for special purpose machines like Terasys and Splash 2. One of the main features of dbC is its ability to express architectural features. Users can define linear arrays or multidimensional mesh structures. The compiler takes care of mapping the user architecture on the target machine.

Languages like VHDL and Verilog can encode parallelism at various levels of hardware design. For ASIC development, designers often use a hardware description language. On Splash 2, VHDL is the only language used for algorithm development.

In programming commercial parallel processors, vendor developed languages have to be used. For example, C* is used for the Connection Machine and MPF and MPC are used for the MasPar family of parallel computers. Many attached vector processors have special library routines that are callable from most high-level languages. Often, for the purpose of programming workstation clusters, a communication library is available, e.g., PVM [71], and Express [96]. Many vision algorithms have been experimented using a workstation cluster using a communication library [117, 186, 183].

2.2 Algorithms

Parallel algorithms for computer vision tasks have interesting communication and computation characteristics. Algorithms for Fast Fourier Transform (FFT), and connected component analysis have become standard text book examples. Parallel algorithms for FFT, connected component analysis and Hough transform have been very widely reported in the literature. In addition to these algorithms, parallel algorithms for vision tasks and related system design issues such as scheduling, and load balancing are also reviewed.

2.2.1 System design issues

Load balancing is an important issue in order to increase the speedup achieved in a distributed memory system. Gerogiannis *et al.* [73] describe a load balancing method for image feature extraction on CM-2 and iPSC/2. Their redistribution algorithm is suitable for applications involving local computations, followed by integration of partial results. The distributed scheduling and resource allocation problems arise when the architecture of the target machine is different from the expected architecture of the algorithm or when there are more virtual processors than the number of physical processors. Chaudhary *et al.* [36] have proposed a mapping scheme by analyzing the communication overheads in mapping a problem graph (parallel algorithm) to a host graph (target architecture). Weil *et al.* [228] describe a dynamic intelligent scheduling and control algorithm for reconfigurable architectures for computer vision and image processing tasks. The dynamic scheduler attempts to balance the overall processing scenario with the needs of the individual routines of the task.

A synchronous model for parallel image processing has been described in [235]. A high-level description of the architectural requirements of the application is analyzed to arrive at the complexity of the components needed. Lee and Aggarwal [135] propose a system design and scheduling strategy for a real-time image processing system by optimizing processing speed and load.

2.2.2 Vision and image processing applications

- FFT: One of the most widely studied image transform is the discrete Fourier transform (DFT). The implementation of DFT is done by the well-known Fast Fourier Transform (FFT) algorithm. In addition to its use in image filtering, FFT is applicable in polynomial multiplication and integer multiplication problems. An $N-$point FFT can be computed in $logN$ steps on a N-node hypercube. Gertner and Rofheart [74] describe a 2D-FFT algorithm with no interprocessor communication. Johnson and Krawitz [113] have presented their implementation on a Connection Machine CM-2. Usually, it is assumed that the number of data points (N) is a power of 2. Swartztrauber *et al.* [211] describe an algorithm for any arbitrary N on a hypercube. A hardware implementation of FFT is presented in [51]. The scalability issues with respect to the number of PEs and communication capability are analyzed by Gupta *et al.* [83]. They conclude that a mesh connected multicomputer does scale as well as a hypercube for FFT implementation. Parallelization of the FFT algorithm on a shared memory MIMD machine is presented in [16]. Approaches to parallel implementation of FFT are presented in most text books on parallel algorithms [108, 137, 190].

- Hough transform: A significant number of researchers have implemented the Hough transform on various architectures. Standard algorithms are described in [137]. The Hough transform is a useful technique for the detection of parametric shapes such as straight lines and circles in images. The parameter space is discretized into bins and votes for each bin are counted to arrive at the dominant parameter set. Ben-Tzvi *et al.* [21] describe an algorithm suitable for distributed memory MIMD architectures and have achieved a real-time performance on a custom-built MIMD machine. Ibrahim *et al.* [97] present their algorithm on a SIMD machine called NON-VON. Using buses as topological descriptors in addition to fast communication and data transfer, Olariu *et al.* [164] present an efficient algorithm for Hough transform. Two ASIC systolic architectures for Hough transform have been designed by Van Swaaij [210]. Jenq and Sahni [112] have developed an $O(plog(N/p))$ algorithm for a reconfigurable mess with buses, where p is the number of quantized angles and $N \times N$ is the size of the image. A VLSI implementation of Hough transform is presented in [191]. A real-time implementation using pipelined processors is described in [88]. Little *et al.* [145] describe an implementation on CM-2. Using the Scan Line Array Processor (SLAP), Fisher and Highnam describe a real-time implementation using only a linear array of processors. A modified Hough transform to check contiguity of a straight line is implemented on a systolic array by Li *et al.* [140]. A generalized VLSI curve detector is presented in [43]. Abbott *et al.* [1] have described an implementation of Hough transform on Splash 2. A summary of different implementations of Hough Transform is shown in Table 2.1.

- <u>Connected Component labeling:</u> Another extensively studied problem in computer vision is the connected component labeling [108, 137]. A connected com-

Algorithm	Description
Ben-Tzvi *et al.* [21]	Custom built MIMD, synchronous multiprocessor
Ibrahim *et al.* [97]	Massively parallel SIMD tree machine called NON-VON
Olariu *et al.* [164]	Bus based reconfigurable mesh
Van Swaaij [210]	Two ASIC systolic architectures
Jenq and Sahni [112]	Reconfigurable meshes with buses
Rhodes *et al.* [191]	A VLSI processor
Hanahara *et al.* [88]	Special purpose processor
Little *et al.* [145]	CM-2
Fisher and Highnam [67]	Scan Line Array Processor (SLAP)
Cheng *et al.* [43]	A VLSI implementation for generalized curve detector
Li *et al.* [140]	A systolic array
Abbott *et al.* [1]	Splash 2 – a custom computing machine
Chung *et al.* [48]	Reconfigurable mesh
Shankar *et al.* [201]	Hypercube using sparse array representation

Table 2.1: Summary of different implementations of Hough transform.

ponent is a maximal-sized connected region where there exists a path between any two pixels in the region. Several algorithms are described for connected component labeling for binary and gray-level images [91]. Embrechts *et al.* [60] describe a MIMD algorithm on an iPSC/2 hypercube. On BLITZEN – a 2-D mesh that allows diagonal transfers, it has been shown to take $O(N^2 logN)$ time steps. Olariu [165] describes an O(logN) algorithm on a reconfigurable mesh. A theoretical analysis on an EREW model is carried out in [54]. A coarse grain approach has been taken by Choudhary and Thakur [46]. Alnuweiri and Prasanna [7] provide a survey of different algorithms for component labeling. A constant time algorithm on a reconfigurable network of processors is presented in [6]. A pyramid algorithm for component labeling is described by Biswas *et al.*

[26] which works for gray scale images also. With a mapping of one processor per pixel on a mesh connected architecture, Hambrusch *et al.* [85] describe two $\theta(n)$ algorithms for an $n \times n$ image on an $n \times n$ mesh-connected computer. For binary images, Manohar *et al.* [148] describe a 3-stage algorithm using a SIMD mesh architecture. Image normalization for translation, rotation and scaling has been attempted by Lee *et al.* [136] on a mesh connected array processor. A VLSI architecture for the same problem has been described by Cheng *et al.* [43].

- Low-level operations: The parallelism being quite visible at the pixel level, many algorithms are available under this category. Kim *et al.* [123] describe implementation of low-level algorithms on a micro-grained array processor (MGAP). Fujita *et al.* [69] describe their IMAP system and report timings of low-level operators on IMAP. Filtering and convolution have been implemented on virtually every parallel processing platform. Both 1-D and 2-D convolutions are popular algorithms studied by many researchers. Non-linear filtering needed in image morphology has been extensively studied [218, 27, 138, 116]. For edge detection using regularization, Poggio [174] proposed a special-purpose parallel hardware. A hardware-based image smoother is presented in [20]. Data replication algorithm proposed by Narayanan *et al.* [159] has been applied to convolution, and histogram computation. Systolic algorithms for digital filters and convolution are presented in [61]. Many low-level tasks including histogram computation, and median filtering on reconfigurable meshes are described in [163]. A fast histogram computation on a reconfigurable mesh is described in [109]. On a SIMD hypercube, efficient histogramming for an $N \times N$ image is possible in $O(logM * logN)$ steps using radix sort, where M is the number of gray levels, and N is the number of PEs [144]. Little *et al.* [145] describe their implementation of low-level algorithms on CM-2. Image shrinking and expanding on a pyramid is described in [111]. A VLSI architecture for obtaining edges using Laplacian of Gaussian is presented in [179]. Image compression and decompression are emerging problems in vision for which many architectures have been presented. Bhama *et al.* [23] describe a parallel implementation of K-L transform for image compression. A VLSI processor based on systolic architecture for image compression using vector quantization has been developed by Fang *et al.* [62]. They report a throughput rate of 25 million pixels per second with an equivalent computing power of 600 million instructions per second. A new architecture for motion-compensated image compression is presented in [237]. Hamdi [86] presents parallel architectures for wavelet transforms.

- Intermediate-level operations: Many intermediate-level tasks such as thinning, segmentation, clustering, image reconstruction and relaxation-based segmentation have been attempted using parallel architectures. Heydon and Weidner [95] describe performance analysis and parallelization of parallel thinning algorithms on Cray supercomputers. On a pyramid architecture, parallel algorithms for medial axis transform have been described in [50]. A VLSI architecture for medial axis transform has been described in [177]. Khotanzad *et al.* [122]

describe a parallel segmentation algorithm on a Sequent computer. A region segmentation using gray level mean difference has also been carried out on Sequent computer [39]. A special-purpose VLSI array processor has been designed by Koufopavlou and Goutis [126] for image reconstruction in tomographic applications. Based on a regular network of cells that can run asynchronously and exchange messages, Lattard and Mazare [131] present a VLSI architecture for image reconstruction. A systolic algorithm for finding k-nearest neighbors is described in [40].

Relaxation is a useful algorithm in computer vision that employs a set of locally interacting parallel processes to update pixel labels in order to achieve a globally consistent interpretation of the image data. Derin and Won [58] describe a VLSI design for image segmentation using relaxation. Gu *et al.* [82] present several VLSI architectures for speeding up the discrete relaxation algorithm. A linear array architecture has been proposed by Chen *et al.* [38] for probabilistic relaxation operations on images. An architecture based on round robin communication between PEs, a parallel architecture for relaxation, has been presented in [120] with applications to character recognition problem. Dixit and Moldovan [59] describe a discrete relaxation technique using a semantic network array processor (SNAP). Data clustering is a compute intensive problem. Li and Fang [141] describe parallel algorithms on SIMD machines. On a hypercube SIMD machine, Zapata *et al.* [238] describe a fuzzy clustering algorithm. Ni and Jain [160] describe a VLSI architecture for pattern clustering. Using cellular algorithms for gap filling, and segment detection, a parallel implementation on a 2-D systolic array is described in [173]. A real-time distance transform processor is described in [236].

- High-level operations: For high-level vision tasks many different approaches have been taken. The high-level tasks are characterized by non-local communication and deal with symbols and strings. Cheng and Fu [41] describe a VLSI architecture for string matching. Three-dimensional object recognition from range images has been implemented on a Butterfly multiprocessor in [24]. Using geometric hashing, object recognition has been implemented on CM-2 by Rigoutsos [192] and on CM-5 by Wang *et al.* [224]. Graph matching has been implemented on MasPar by Allen *et al.* [5]. Motion analysis is a compute intensive job. A VLSI architecture for dynamic scene analysis is described in [178]. Cheng *et al.* [42] present a VLSI design for hierarchical scene matching. Iconic indexing has been implemented on CM-2 using mesh and pyramid algorithms [45]. Parallel algorithms for hidden Markov model on the Orthogonal Multiprocessor has been described in [133]. A real-time face recognition system using a custom VLSI hardware is developed by Gilbert and Yang [76]. Tanimoto and Kent [213] describe special architectures and algorithms for image-to-symbol transformations.

- Neural Networks: Over the last decade many special architectures and VLSI designs have been proposed for implementing artificial neural networks (ANNs). The research work in this area can be classified into 3 categories of architectures:

(i) using existing parallel processors and DSPs, (ii) design of special purpose VLSI chips, and (iii) design of analog and mixed (analog and digital) architectures. There are six types of parallelism available in an ANN [161]. Out of them, node-level and weight-level parallelism are frequently exploited. Ghosh and Hwang [75] investigate architectural requirements for simulating ANNs using massively parallel multiprocessors. They propose a model for mapping neural networks onto message passing multicomputers. Liu [146] presents an efficient implementation of backpropogation algorithm on the CM-5 that avoids explicit message passing. The results of CM-5 implementation has been compared with results on Cray-2, Cray X-MP and Cray Y-MP. Chinn *et al.* [44] describe a systolic algorithm for ANN on MasPar-1 using a 2-D systolic array-based design. Onuki *et al.* [166] present a parallel implementation using a set of sixteen standard 24-bit DSPs connected in a hypercube. Kirsanov [124] discusses a new architecture for ANNs using Transputers. A multi-layer perceptron implementation has been described in [161] using GAPP - a systolic array processor chip. Muller [158] presents a special purpose parallel computer using a large number of Motorola floating point processors for ANN implementation. Architecture of SNAP-64 – a 1-dimensional ring of parallel floating point processors is described in [153].

Several special-purpose VLSI chips have been designed and fabricated for ANN implementation. Sato *et al.* [198] describe a 64-neuron chip. A neurocomputer consisting of 512 neurons has been built and shown to be six times faster than Hitachi-280 supercomputer. Connectionist network supercomputer (CNS-1) uses a RISC CPU with a vector processor as the building block in a 128 PE CNS-1 [12]. Direct emulation up to a fixed size of nodes and virtual emulation beyond that size is supported in a digital neurocomputer based on a special chip supporting four neurons per chip in the design of Pechanek *et al.* [171]. The system has been evaluated for NETTALK emulation and shown to be approximately 43 times faster than an implementation of NETTALK on CM-2. Using 64 processing nodes per chip and hardware-based multiply and accumulate operators, a high performance and low cost ANN is presented by Hamerstorm [87]. A binary tree adder following parallel multipliers are used in SPIN-L architecture proposed by Barber [19]. Shinokawa *et al.* [203] describe a fast ANN (billion connections per second) using 50 ASIC VLSI chips. Using a 2×2 systolic PE blocks, MANTRA-I neurocomputer is described by Viredez [223]. A tree of connection units with processing units at the leaf nodes has been proposed by Kotolainen *et al.* [125] for mapping many common ANNs. Asanovic *et al.* [13] have proposed a VLIW of 128-bit instruction width and a 7-stage pipelined processor with 8 processors per chip. Ramacher [176] describes the architecture of SYNAPSE – a systolic neural signal processor using a 2-D array of systolic elements.

Several stochastic neural architectures have been described in [55, 167, 216]. The main advantage of this approach is that there is no need for a time consuming and area costly floating point multiplier which makes them very suitable for

VLSI implementations. Using a mixed design involving both analog and digital architecture, Masa *et al.* [152] describe an ANN with a single output, six hidden layers and seventy inputs that can perform at 50 MHz input rate. There have been many other well-known neural network chips and architectures e.g., ANNA from AT&T [195], CNAPS [161] and GANGLION[52].

Another line of effort is concerned with parallel implementation of image processing tasks on images represented by special data structures. Encoding binary images as bit strings and processing these bit streams in parallel is described by Wu *et al.* [233]. Processing on images represented by quadtrees and region boundaries is presented in [25, 201, 232].

2.3 Special-purpose hardware

Many vision systems have been built around specific processors. Recently, MVP-80 – an advanced digital signal processor (DSP) supporting multiprocessing has been shown to be very effective for developing many vision systems. This section starts with a brief description of MVP-80.

2.3.1 Multimedia Video Processor (MVP-80)

MVP-80, designed and developed by Texas Instruments, is one of the most advanced DSPs supporting multiprocessing. This special processor has been used for many vision applications, including video conferencing, document image processing, graphics, image compression, image tracking and diagnostic imaging. The architecture of

MVP-80 is shown in Figure 2.1. It contains five processors (4 DSPs and one 32-bit RISC CPU) in a single silicon wafer. The five processors can execute independently and concurrently. Each processor has its own private memory. The processors can access other parts of memory through a crossbar concurrently. High throughput is achieved by running all the five processors accessing their 2KB private cache (4KB for the RISC processor). The RISC processor has a floating-point unit. The on-chip video controller can display/capture video data. The DSP CPUs have a 64-bit instruction width and 32-bit address/data width. Two independent address buses are available on each DSP. Each DSP has a 32-bit ALU and a 16-bit multiplier. The ALU can be configured as two independent 16-bit ALUs or as four 8-bit ALUs. Similarly, the 16-bit multiplier can also act as two 8-bit multipliers. The RISC processor has a 3-stage pipeline. In addition, there is an on-chip memory controller, called the Transfer Controller (TC) to interface with external memory. The floating point unit has a peak performance of 100 MFLOPS and so a single MVP-80 can deliver up to 2 billion Operations (BOPS) per second. MVP-80 can be programmed using a special C-compiler or Assembler/Linker. The best performance has been obtained by programming a MVP-80 with the Algebraic Assembly Language [214].

Benchmark results as reported by the manufacturer are shown in Table 2.2. The timings have been scaled up from a 30 MHz MVP-80 to 50 MHz. The following assumptions have been made while reporting the benchmark timings: (i) the entire image is available in the memory, (ii) the 4 parallel processors are being fully utilized, and (iii) the RISC processor is not being utilized.

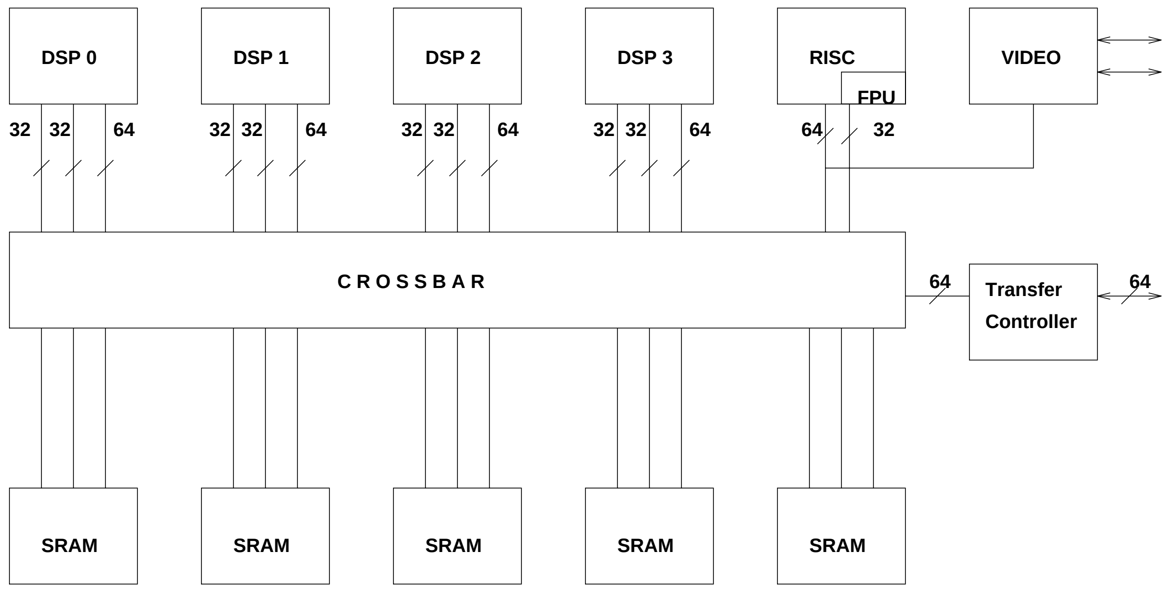


Figure 2.1: Architecture of MVP-80.

Operation	Speed
3×3 median filtering	25 MHz
3×3 convolution (16×16 multiply)	22 MHz
3×3 convolution (8×8 multiply)	40 MHz

Table 2.2: MVP-80 benchmark results.

2.3.2 WARP and iWARP Processors

WARP and iWARP processors were designed and developed at Carnegie-Mellon University (CMU)[53]. Both are examples of systolic array-based architectures for scientific and image processing applications. As a linearly connected set of cells with two communication channels between cells, WARP can deliver up to 10 MFLOPs per cell. Each cell has its own program sequencer and memory. This processor is highly suitable for low and intermediate level vision algorithms. iWARP is an extension of WARP with more powerful individual cells. It also supports a 2-D systolic array. Both WARP and iWARP have been used for stereo vision for obstacle avoidance and color-based road following systems developed at CMU. The tasks need about 1000 operations per pixel and need to process several frames per second. For this application, approximately 100 MFLOPS of computing power has been supported by WARP.

2.3.3 NETRA

Designed and developed at the University of Illinois, Urbana, NETRA is a recursively defined hierarchical multiprocessor system [47]. It supports both distributed as well as shared memory. It has two kinds of processors: (i) processing elements in clusters; and (ii) distributing and scheduling processors (DSPs). The PEs carry out the computations whereas DSPs distribute and control the tasks. The memory subsystem has a shared global memory and a global interconnect network to link PEs and DSPs to the global memory. The schematic block diagram of NETRA is shown in Figure 2.2. The clusters of PEs can operate in either SIMD or MIMD mode. Each

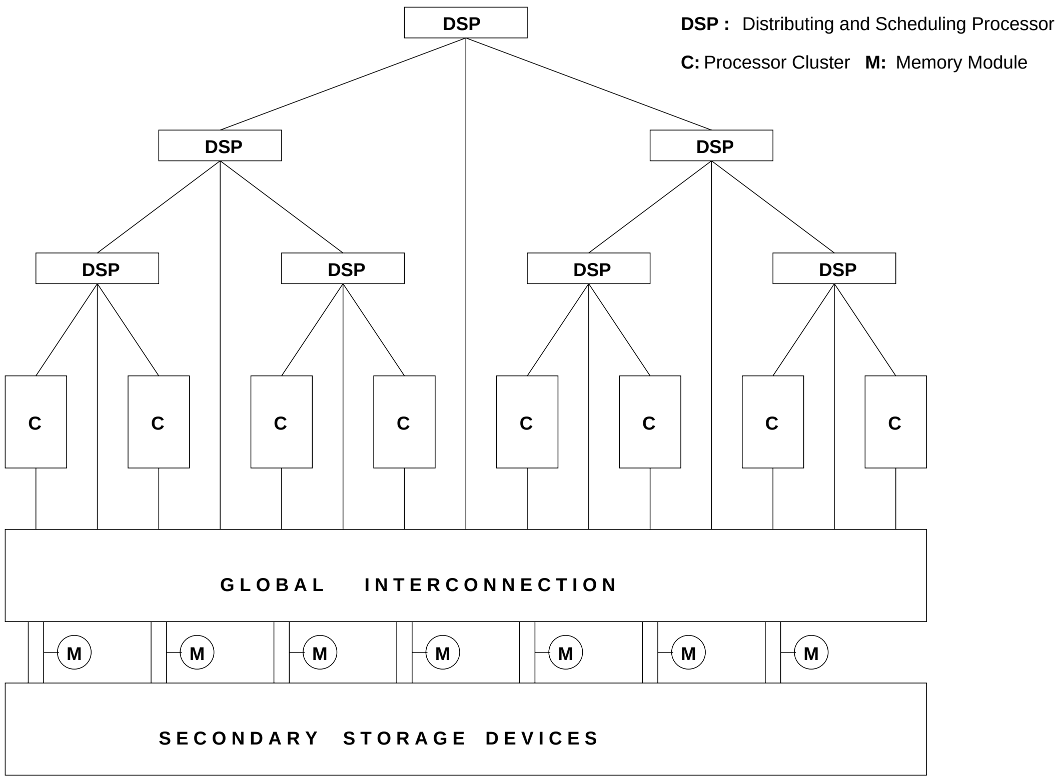

DSP : Distributing and Scheduling Processor
C: Processor Cluster M: Memory Module
DSP
C
M
GLOBAL INTERCONNECTION
SECONDARY STORAGE DEVICES

Figure 2.2: Architecture of NETRA.

processor in the cluster is a general purpose processor with floating point capability. The PEs in a cluster share a common memory. The PEs and the DSPs are connected through a crossbar switch. More details are available in [47]. For a 256×256 image, convolution with a 10×10 mask takes 176msecs on a 16 processor system. On a 32 processor system, 3×3 Sobel operation takes 0.13 secs. For a 5×5 median filter, the performance of a 32 processor system is 1.9 secs.

2.3.4 Image Understanding Architecture (IUA)

In order to meet the computational requirements of all the three levels of vision algorithms, IUA provides three different levels of architectures [226]. The three levels communicate via parallel data control paths. At the lowest level, there is a collec-

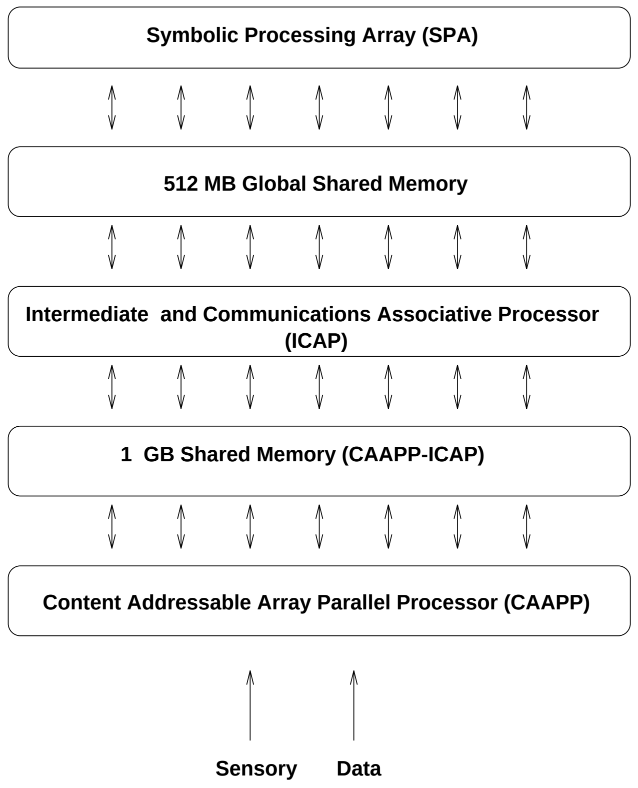

Symbolic Processing Array (SPA)
512 MB Global Shared Memory
Intermediate and Communications Associative Processor (ICAP)
1 GB Shared Memory (CAAPP-ICAP)
Content Addressable Array Parallel Processor (CAAPP)
Sensory
Data

Figure 2.3: Schematic overview of IUA.

tion of 512×512 1-bit serial processors, called Content Addressable Array Parallel Processor (CAAPP). The intermediate level is an array of 64×64 16-bit processors and is known as Intermediate Communications Associative Processor (ICAP). At the highest level there are 64 processors capable of running LISP programs. The highest level is called Symbolic Processing Array (SPA) which also directs the array control unit (ACU). A schematic adapted from Weems *et al.* [226] is shown in Figure 2.3. On a 512×512 grayscale image, a 11×11 convolution only takes 0.2msecs. Connected component analysis on a binary image of same size is performed in 0.07 msecs. More details are available in [226].

2.3.5 An Integrated Vision Tri-Architecture System (VisTA)

Similar to the IUA processor, ViSTA has three distinct layers of processors [209]. At the lowest level is ViSTA/1 which is based on a massively parallel sliding memory plane SIMD processor with a support for low-level vision algorithms. Typically, there are N^2 PEs at this level. Each PE can communicate to any other PE at this level without interrupting that PE. I/O and computation can occur simultaneously. A PE consists of ALU, registers, MUX, DMUX and a switching element. The second layer supports intermediate vision tasks. It consists of N PEs with N memory blocks connected through a communication bus. This layer is based on flexibly coupled multiprocessors [208]. The top most layer is a tightly coupled hypercube multiprocessor. Estimated time for 2-D convolution with a 3×3 mask is $8.8 \mu secs$ and Sobel (3×3) is 2 $\mu secs$. However, a 3×3 median is estimated to take 11.2 $\mu secs$. More details of this system are available in [209].

2.3.6 Scan Line Array Processor (SLAP)

SLAP is a SIMD linear array for real-time image processing [68]. A system contains a controller that selects and orders execution of modules on the sequence vector. A sequencer and a PE vector operate in a lock step mode. The block diagram is shown in Figure 2.4. The sequencer can broadcast messages to all the PEs. It also includes registers that function as virtual PEs, neighbors to real PEs on the vector extremes. It also has a global synchronization line. The PEs in the third stage are 3-stage pipelined processors using custom VLSI processors. SLAP's performance has

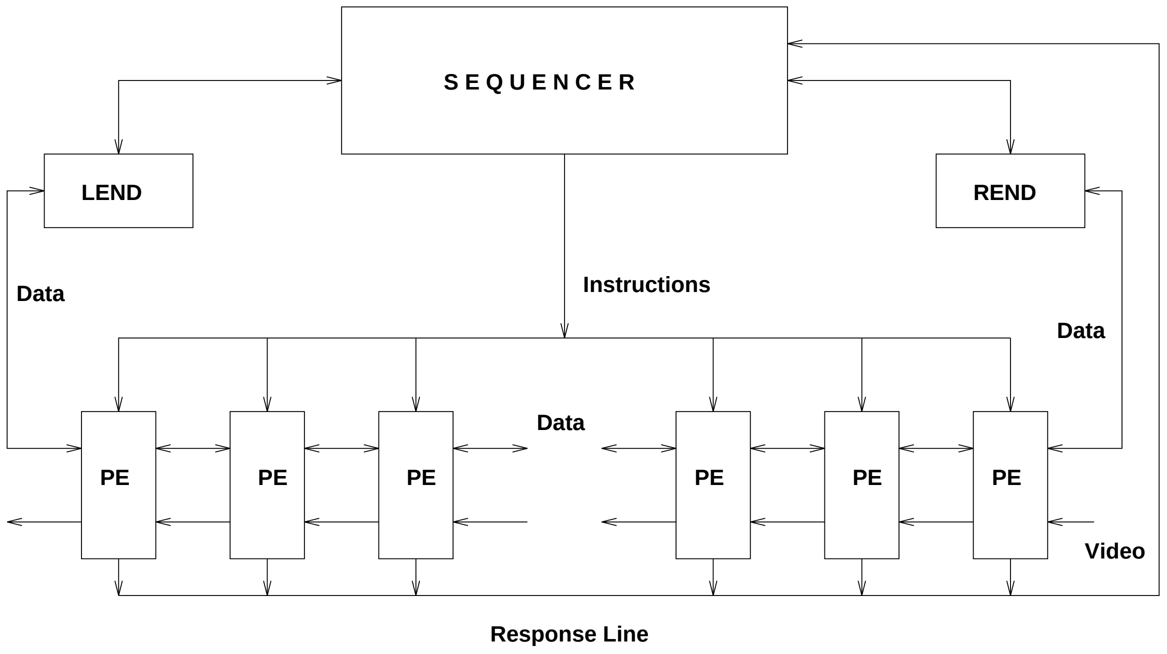


Figure 2.4: Architecture of SLAP.

been compared to a Cray for convolution, median filtering and Hough transform with significant speedups. More details are available in [68, 93].

2.4 Commercial image processing accelerators

Several single and multi-board accelerators are used for high performance image processing. In this section, a brief description of few popular systems is given. Typically, these systems cater to only low-level processing and have dedicated hardware designs for commonly used low-level operators such as point operations and convolution. Often, they are designed to deliver real-time frame rate performance. Compared to the general-purpose parallel processing systems, these systems are relatively cheaper.

2.4.1 PIPE

Pipelined Image Processing Engine (PIPE) is optimized to perform local neighborhood operations on iconic (intrinsic) images [121]. It provides easy multistage, parallel image processing. PIPE system consists of a sequence of identical processors sandwiched between a special input processor and a special output processor. There can be several identical processing stages in between the input and output stages (see Figure 2.5). Each of the processing stages performs a different operation on the image sequence. Each stage receives three input streams and produces three output streams. The three input streams are: (i) from previous stage output stream, (ii) from the output of the present processor, and (iii) from the output of the next processor in the pipeline. Similarly, the three output streams are: (i) to the next stage in the pipeline, (ii) to the previous processor in the pipeline and (iii) to itself. In addition, there are four wildcard paths available for both input and output. The three input images can be weighted and combined in any fashion before they are processed by the processing stage. Two kinds of processing are carried out in each processing stage: (i) pointwise arithmetic and Boolean operations, and (ii) neighborhood operations. Two neighborhood operations can be performed on an image at frame rate (30 frames/second). Provisions exist to define ‘regions of interest’. Different operations can be applied within each region of interest. The forward and backward paths in the processing stage can be effectively used to build image pyramids. It is designed for low-level processing. Commercial systems from ASPEX and Datacube are similar to the PIPE design philosophy. More details about PIPE system are available in [121].

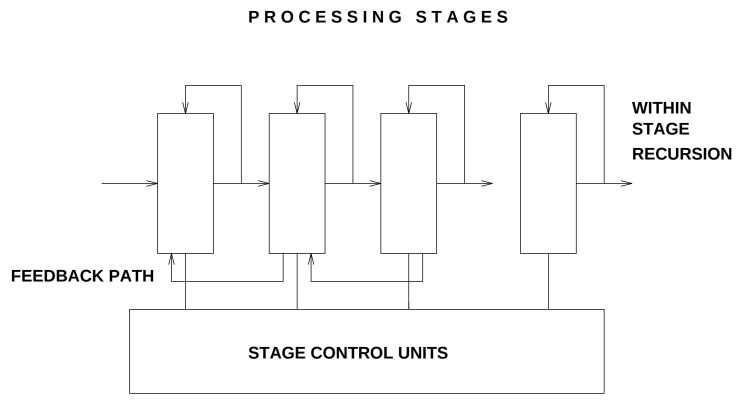


Figure 2.5: Schematic of a PIPE processor.

2.4.2 Datacube MV-250

One of the most popular high-end image processing accelerator is Datacube's MaxVideo 250 or MV-250. Improving upon its predecessor MV-200, it uses a single VME slot. Based on a 20 MHz pipeline, MV-250 consists of several modules as shown in Figure 2.6. In addition to a bus interface module and input/output modules, there are two special processing stages, namely (i) arithmetic unit (AU) and (ii) advanced processor (AP). AU aids linear and non-linear operations on pixels with the help of a custom ALU, four 11-bit multipliers, seven 10-bit ALUs, two run length encoders and two row and column address generators. The role of AP is to support neighborhood operations such as convolution and binary morphology. Support for a 64, one-dimensional 8-bit FIR filter that can be configured as a single 8×8 or two 8×4 kernels is available using the AP. On-board processing for binary morphology is supported for 3×3 structuring elements. A 32×32 8-bit crossbar is available to switch data inputs at video rate from and to computing resources available on the system.

An important aspect of MV-250 is its software environment, called 'maxflow'. Maxflow consists of a set of C-callable routines for connecting different stages of a

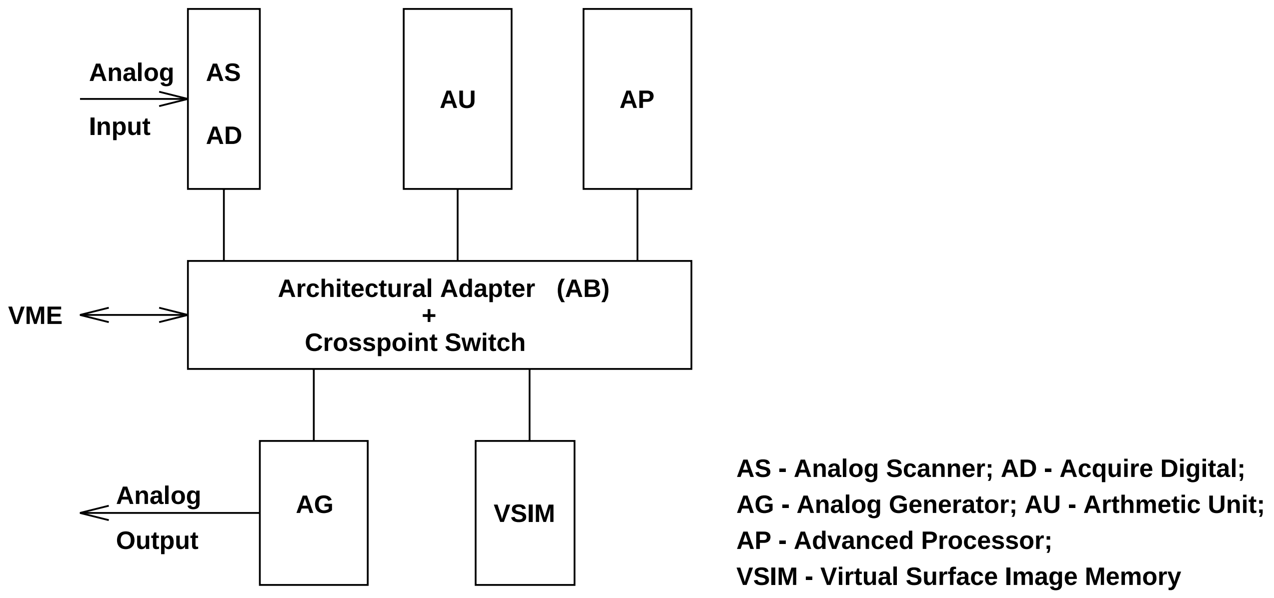

Analog
Input
AS
AD
AU
AP
VME
Architectural Adapter (AB)
+
Crosspoint Switch
Analog
Output
AG
VSIM
AS - Analog Scanner; AD - Acquire Digital;
AG - Analog Generator; AU - Arthmetic Unit;
AP - Advanced Processor;
VSIM - Virtual Surface Image Memory

Figure 2.6: Schematic of Datacube MV-250.

user-defined pipeline, controlling attributes of the processing stages (e.g., kernels for convolution) and to control the overall interaction with the host. A X-window-based GUI is provided to generate Imageflow code automatically. A convolution using 8×8 kernel on a 512×512 image takes approximately 13 msecs. More details of the Max Video systems are available in [57].

2.4.3 Imaging Technology MVC-150

MVC-150 from Imaging Technology is a high performance pipelined image processing accelerator. With a basic pipeline speed of 40 MHz, a 512×512 frame processing can take 7.5 msecs. Many processing stages can be added in the basic pipeline.

A typical system consists of three modules. Extra memory modules can be supported by extending the 24-bit internal bus. The Image Manager (IM) interfaces with external host through a VME bus. The three modules are acquisition module

(AM), computation module (CM) and display manager (DM). The CM can perform convolutions, inter-image arithmetic, feature extraction, morphological operations, object labeling and binary correlation. An advanced processing unit based on TI DSP TMS320C31 is also available. Many of the functions described earlier are carried out on a special-purpose hardware.

The hardware is supported by a multi-layered software development environment. A window based automatic code generation facility also exists. Using the base hardware, a 4×4 convolution can be performed at 40 MHz and 8×8 convolution at 10 MHz. More details are available in [101].

2.4.4 Alacron i860 and Sharc multiprocessor boards

For defense and satellite image processing applications, Alacron has several high performance accelerators on PCs as well as VME-based systems. These multiprocessor pipelined systems are based on either Intel i-860 or Analog Devices Sharc 21062 DSPs. The multiprocessors are connected on a cluster or grid in the multi-Sharc systems. The multi-processor systems can work in SIMD or MIMD modes. The system has an Intel $i960$ as the control processor. The $i860$ based systems have two $i860$ processors. These systems are programmable through a set of C-callable routines. Very high performance of the order of Giga operations per second have been reported for the Sharc-based system. On the $i860$-based systems, a 3×3 convolution takes 51 msecs. More details are available in [4].

2.4.5 Data Translation DT-2867

At the low-end, there are many PC-based image processing accelerators that can be interfaced to frame grabbers for direct data input. Data Translation's DT-2867 is a popular frame processor. The processor board has three ALUs, three 16-bit fixed point multipliers, one divider, histogram generator and several lookup tables. Using these hardware resources, real-time frame averaging, frame arithmetic and logic operations and histogram generation can be performed. 3×3 convolution and morphological operations take close to two frame periods, i.e., 15 msecs. Several dedicated hardware blocks are used to realize the functions. A high level language callable set of routines are provided to interface the special hardware in applications. More details are available in [56].

2.5 General-purpose parallel processors

Many general-purpose parallel processing systems are being used for computer vision tasks. Thinking machine corporation's CM-5, IBM's SP-2 and MasPar's MP-2 are the three systems briefly presented in this section. Other general-purpose parallel processors such as Intel's Paragon and BBN Butterfly have also been used for vision tasks.

2.5.1 SP-2

IBM's *powerparallel* SP-2 is a scalable parallel system which is being used for a wide range of applications. The delivered power of SP-2 has been shown to be in the range of Gigaflops. SP-2 is based on a distributed memory, message passing architecture. A SP-2 can consist of up to 512 processing elements based on RISC System/6000. The PEs are connected by a high performance, multistage, packet-switched network for interprocess communication. Each node can run IBMs AIX operating system and associated applications. Many special-purpose software packages are available to exploit the power of a full-blown SP-2. Important design issues of SP-2 are explained in [3], and the main design issues of SP-2 are given below.

- A high performance scalable parallel system must utilize standard microprocessors.
- Small latency and high bandwidth for inter-process communication will demand a custom interconnect network.
- System must provide high performance parallel libraries, parallel file system, parallel I/O facilities and state-of-the-art execution support.
- Very fast recovery from single point failures to increase the system availability should be supported.

The architecture of SP-2 is shown in Figure 2.7. Many standard benchmarks have been run on SP-2 [2]. Judd *et al.* reported high performance for pattern clustering using the earlier model SP-1 [117]. On SP-2, Chung *et al.* [49] reported twice the performance of a CM-5 for linear feature extraction.

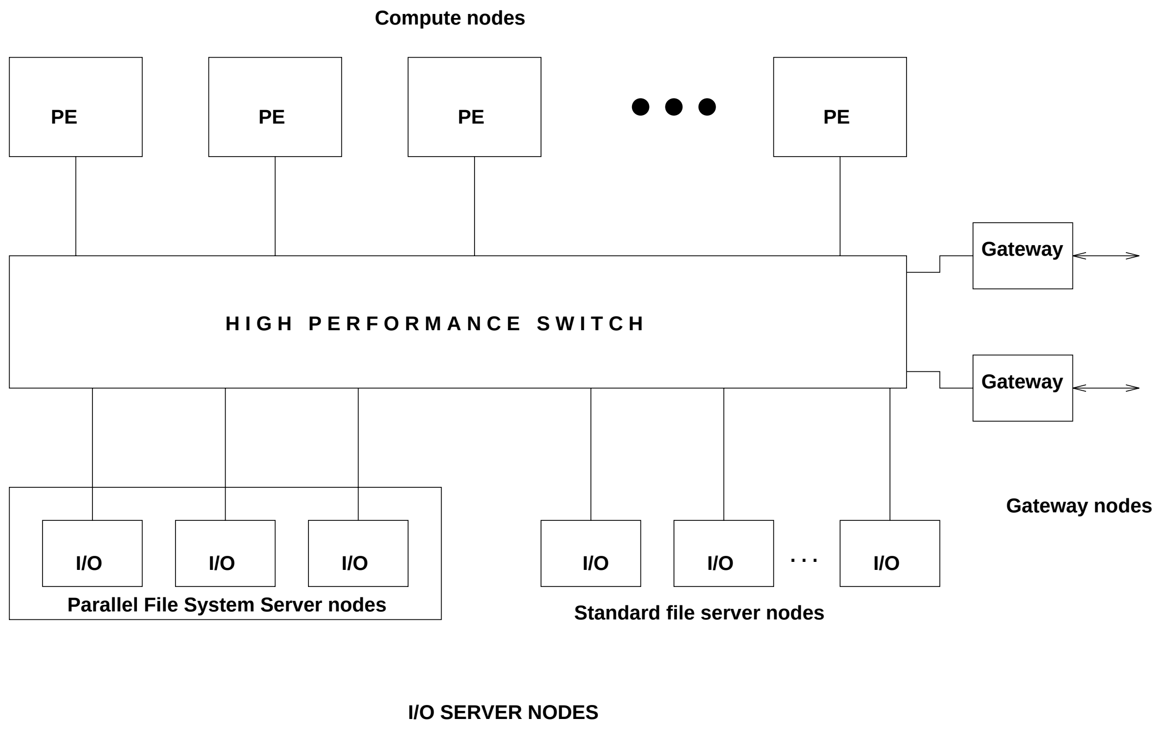


Figure 2.7: Architecture of SP-2.

.

2.5.2 CM-5

Overcoming the rigid SIMD architecture of the CM-2, Thinking Machines Corporation designed CM-5 that combines the advantages of both SIMD and MIMD processing. The CM-5 uses synchronized MIMD processing to provide a good performance for both synchronized communication and branching.

The architecture of CM-5 is shown in Figure 2.8. A CM-5 can consist of 32 to 16,384 processing nodes and a vector processing unit. There can be several sequencers. Both the sequencer and the PEs are SPARC-based processors. The I/O subsystem consists of mass storage devices and network interfaces. These three types of building blocks are interconnected by three networks, namely, (i) data network, (ii) control network, and (iii) diagnostic network. Point-to-point communication is supported

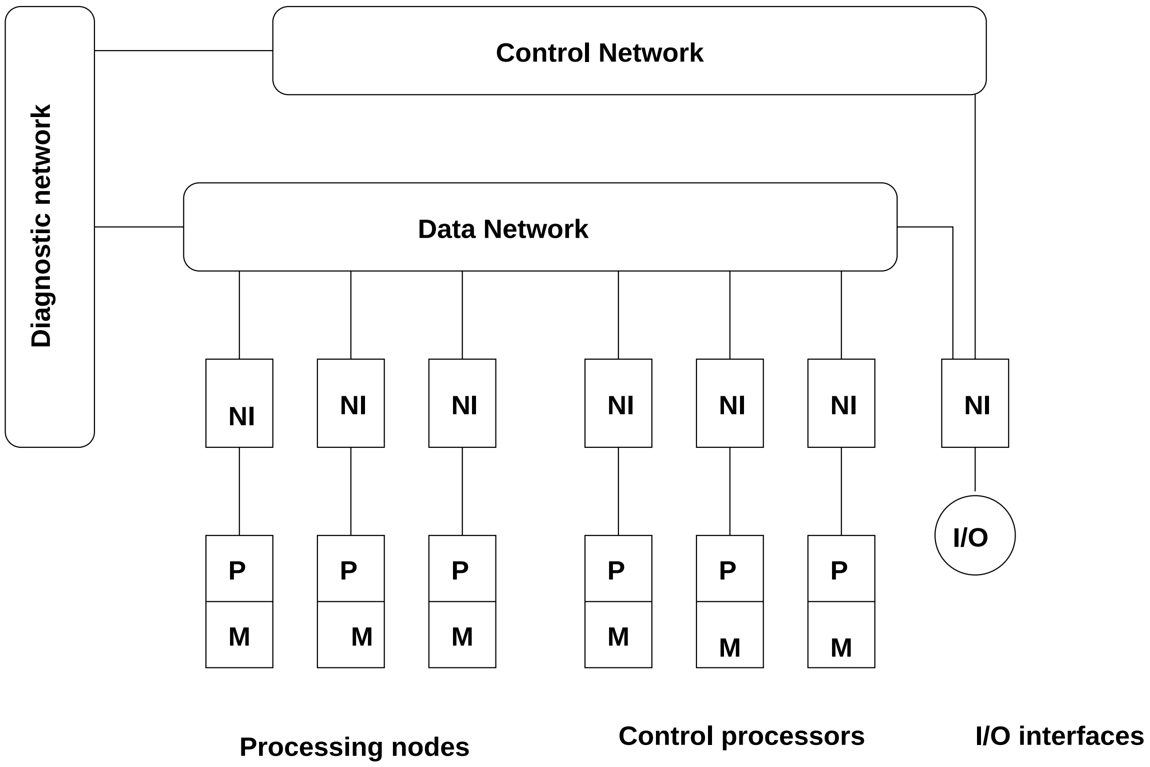


Figure 2.8: Architecture of CM-5. (Adapted from [96])

by the data network. Broadcast, synchronization and scan operators are supported by the control network. The diagnostic network supports on-line diagnostics for the overall system. The system has a peak performance of 1 Tflops. More details are available in [96].

2.5.3 MasPar MP-2

The MasPar family of supercomputers consists of massively parallel SIMD machines with up to 16K PEs operating synchronously on multiple data elements. An Array Control Unit (ACU) sends out a stream of instructions to all the PEs. A PE can either execute the instruction or decide to be passive. Each PE has a private memory of 64K bytes. Interprocess communication is carried out in three ways: (i) broadcast

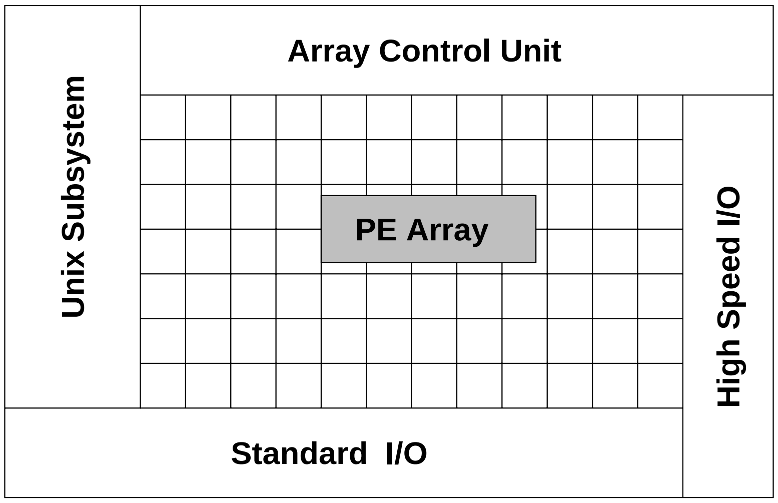


Figure 2.9: Architecture of MP-2.

by ACU, (ii) nearest neighbor communications through Xnet and (iii) through global router. A floating point unit and an integer arithmetic unit is shared between the processor cluster. Because of its fine grained SIMD architecture, it is suitable for many low-level operations. A schematic of MP-2 is shown in Figure 2.9.

2.6 Summary

A brief summary of work in the area of parallel architectures and algorithms for computer vision is given in this chapter. Many special-purpose hardware and general-purpose parallel processors have been reviewed. Parallel algorithms for many vision tasks from all the three levels of vision task hierarchy have been surveyed. For their interesting communication and computational characteristics as well as their general utility, algorithms for FFT, connected component analysis and Hough transform have been widely studied. Parallel languages specific to vision have not received adequate attention.

Chapter 3

Custom Computing Machines

Once an appropriate computer vision algorithm has been selected for a given task, it needs to be implemented in an efficient way to meet the desired response time requirements. An application-specific hardware design, such as application specific integrated circuit (ASIC), is the best implementation to provide the fastest execution time. However, there are many limitations in designing an ASIC. The most important limiting factor is the development cost and time. For low-volume applications, the fixed costs associated with designing an ASIC are very high. Further, once an ASIC has been designed and fabricated, it is very difficult to make modifications or corrections to the design. The motivation for reconfigurable hardware comes from its ability to overcome these limitations. In this chapter, the current hardware implementation technologies with specific reference to field-programmable gate arrays (FPGAs) will be presented. The custom computing paradigm is based on the ability of an architecture to tailor itself to the application needs. The FPGAs provide a suitable hardware platform for custom computing by virtue of their field programmability.

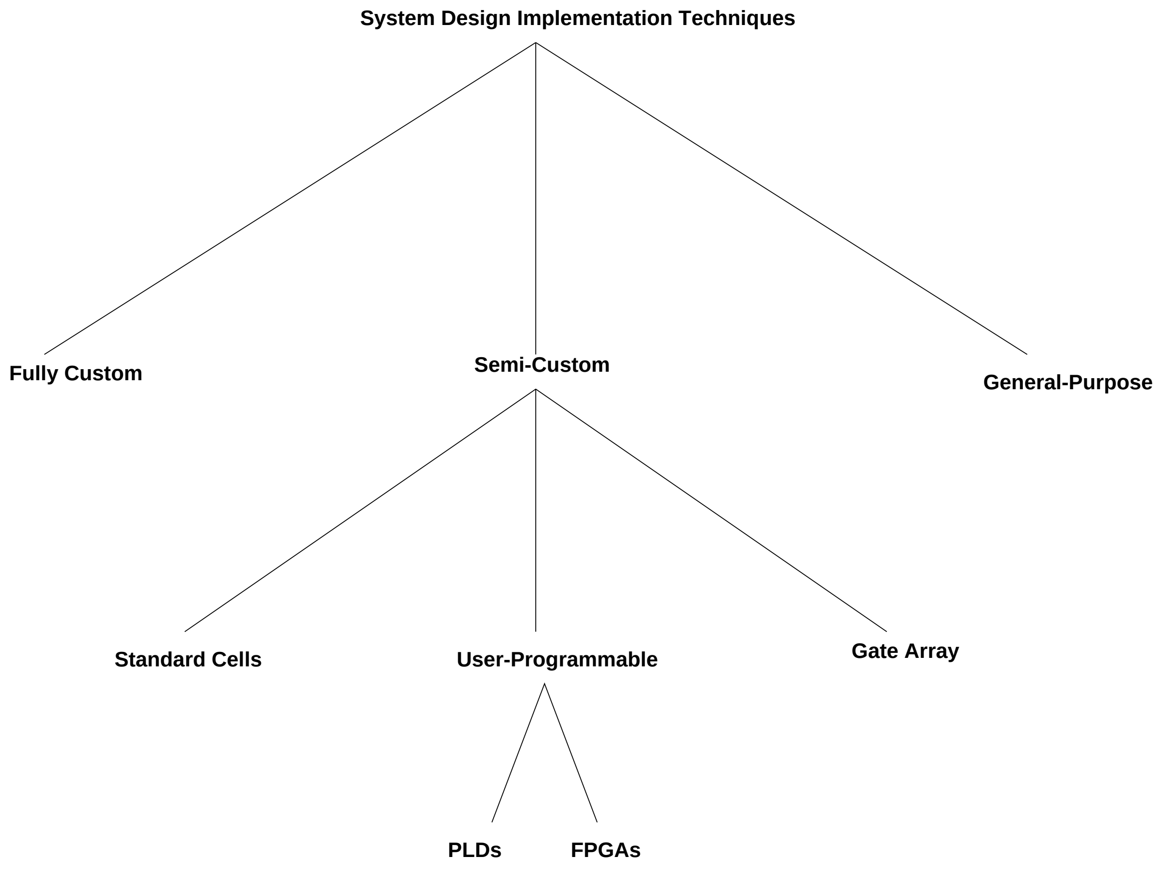


Figure 3.1: Methods for embedded system design.

3.1 Field programmable gate arrays (FPGAs)

There are many methods for implementing an embedded system for computer vision. The available options are shown in Figure 3.1. The fully-custom method is cost effective for high-volume applications. The semi-custom approach, though slower than the fully-custom method, offers cost-effective solutions. In the semi-custom design category, field-programmable gate arrays are of special interest as they offer many advantages over the standard cells or gate-arrays. FPGAs consist of electrically programmable gate arrays whose integration capabilities are like mask-programmable gate arrays (MPGA). They are also similar to PLA-based programmable logic de-

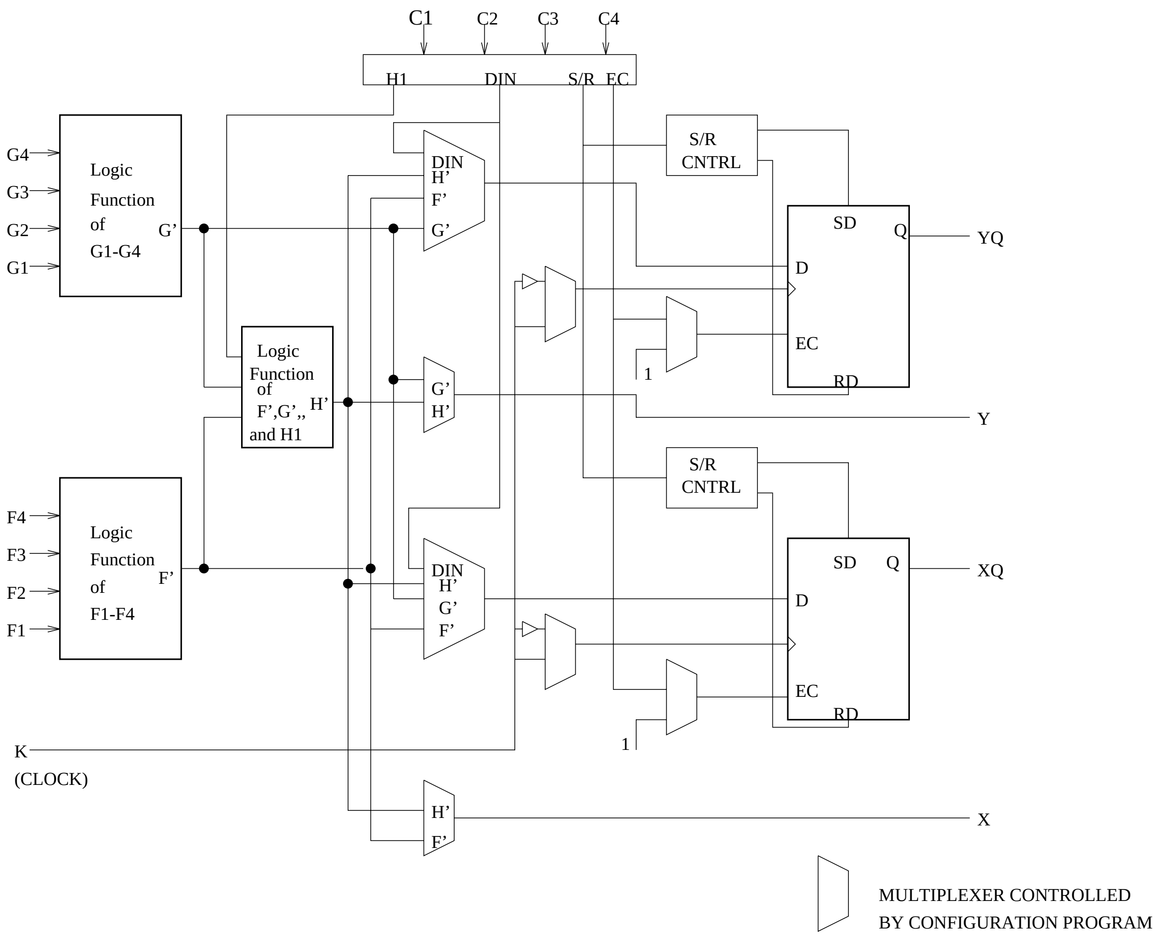


Figure 3.2: Structure of a Xilinx 4010 CLB.

vices in terms of rapid development time. Sequential and combinational logic can be implemented using multi-stage approach. Architecturally, an FPGA is characterized by three types of building blocks, namely, (i) Configurable Logic Blocks (CLBs), (ii) Input/Output Blocks (IOBs), and (iii) Interconnection Networks. The structure of a CLB can be as simple as a transistor to as complex as a microprocessors [193]. The CLBs can be arranged in a row, or, more commonly, in a matrix form. A typical CLB of Xilinx 4000 series FPGA is shown in Figure 3.2. The total number of CLBs on a FPGA also varies from vendor to vendor. In Xilinx 4010, there are 400 CLBs. The IOBs provide an interaction with the external world. The most space consuming component on a FPGA is the interconnection network which supports interconnections between CLBs to logic synthesis. Several different programming methodologies are used in the interconnection network. The three commonly used programming methods are:

1. SRAM-based: a 'pass' transistor connects two inputs if the SRAM bit is 'ON',
2. Antifuse-based: a fuse once blown can permanently connect two inputs, and
3. EPROM-based: a floating gate can connect two inputs based on the gate current.

Commercially available FPGAs differ on the basis of: (i) CLB architecture, (ii) number, size and capability of CLBs, (iii) number of IOBs, and (iv) programming methodology. A survey of commercially available FPGAs is shown in Table 3.1. FPGAs have been a topic of special interest because they support user logic programmability without compromising speed and flexibility which is also exploited in building custom computing machines.

The design flow in programming FPGAs starts with the entry of logic function by logic expressions, schematics or high-level hardware description languages (HDLs). A *netlist* is generated from the input specification before logic partitioning is carried out. Computer Aided Design (CAD) tools are used to place and route the logic on a FPGA. This style of programming a FPGA makes it suitable for tailoring special purpose architectures. The designer can verify the logic using logic simulators. Using the synthesis tools, the designer can determine whether the logic can be accommodated on a given FPGA. The delays involved in the interconnection network can be used

Manufacturer	General Architecture	Logic Block type	Programming Methodology
Xilinx	Symmetrical Array	Look-up Table	Static RAM
Actel	Row-based	Multiplexer-based	Antifuse
Altera	Hierarchical PLD	PLD Block	EPROM
Plessey	Sea-of-gates	NAND Gate	Static RAM
Plus	Hierarchical PLD	PLD Block	EPROM
AMD	Hierarchical PLD	PLD Block	EEPROM
QuickLogic	Symm. Array	Multiplexer based	Antifuse
Algotronix	Sea-of-gates	Multiplexers and gates	Static RAM
Concurrent	Sea-of-gates	MUX and gates	Static RAM
Crosspoint	Row-based	Transistor pairs	Antifuse
AT & T ORCA	Array	Look-up Table	Static RAM
Motorola MPA 1000	Array	Look-up Table	Static RAM

Table 3.1: Commercially available FPGAs.

to estimate the peak speed for the logic. A new algorithm can be implemented on the same FPGA assuming that it supports reprogrammability (except in the case of antifuse-based FPGAs) by changing the control bit stream.

Xilinx 4010 has 10,000 usable gates. To program the 400 CLBs, a control bit stream consisting of 178,096 bits is required. This bit stream is loaded every time in terms of three parts (i) preamble, (ii) 788 frames of 226 bits each and (iii) postamble frame. For more details, refer to [234].

The power consumption of these devices vary as per the CLB utilization and the clock rate. At the maximum, each device dissipates 2.8 Watts although the actual power dissipated is far less. Details of computing the power dissipation for each device is available in [234].

3.2 Recent trends in FPGAs

Since their introduction in 1984, FPGAs are undergoing numerous technological advances. Some of the advances are in the directions of increased logic capacity, more advanced features for supporting coprocessing and better routing technology. Most SRAM based FPGAs now support dynamic partial reconfigurability. Recently, Xilinx has announced its XC6200 series that supports coprocessing needs such as CPU readable on-chip memory/registers, faster as well as partial reconfigurability and user defined memory/logic allocation. Xilinx's XC5200 is yet another powerful series of SRAM-based FPGAs. In the XC5200 series, the CLBs are themselves a collection of four logic cells, where each logic cell consists of a 4-input function generator, a storage device and control logic. The local routing resources have been combined with logic resources to form a 'VersaBlock'. General purpose routers connect to the VersaBlock through general routing matrix. In addition, there is a 'VersaRing' aound the chip enclosing all the CLBs. The abundance of routing resources help the design automation tools while placing and routing the logic. The gate count per FPGA is also on the rise. Already, 100K gate FPGAs are being targeted (e.g., XC6264). These enhanced features will certainly have a positive impact on the future of the CCMs. In the area of one-time programmable FPGAs, recently both Xilinx and QuickLogic have announced a new technology for interconnection that uses a combination of SRAM and antifuse technology. The main advantage is that these FPGAs can be 100% utilized with guaranteed response time. However, these are not good for designing CCMs.

3.3 Survey of FPGA-based computing machines

A *custom computing machine* (CCM) is an embedded system that can be used to build specialized architectures for different applications. FPGAs have revolutionalized custom computing by virtue of their user reprogrammability. Many experimental and commercial FPGA-based computing machines have been reported in the literature. BORG and BORG II [34] are based on Xilinx FPGAs running as an attached processor on PCs. A neural network based pattern classifier based on Xilinx 3090 FPGAs is reported in [52]. Researchers at Brown University have developed PRISM [15]. In the commercial category, "The Virtual Computer" based on Xilinx 4013 has been developed by Virtual Computer Corporation. Digital Equipment Corporation has developed "4Mint" based on Xilinx 4010s. One of the largest CCMs built so far is Teramac, developed by Hewlett Packard [8]. Teramac is based on custom FPGAs packaged in large multi-chip modules (MCMs). A fully configured Teramac has 500 MB of RAM and hardware support for large multiported registers. A comparative analysis of some of the recent custom computing machines is given in Table 3.2. Many more such machines have been reported in the proceedings of the "FPGAs for Custom Computing Machines" workshop [98, 99, 100].

Product name	Designer	Typical system configuration	Comments
Adaptive Connectionist Model Emulator (ACME)	Univ. of California, Santa Cruz	14 Xilinx 4010, 6 Xilinx 3195	3195s are used as programmable interconnect; 4K dual ported memory on each 4010
Configurable Hardware Algorithm Mappable Processor (CHAMP)	Lockheed Sanders, Nashua	16 Xilinx 4013; 512K dual ported memory; Crossbar using FPGAs	Hardware prototyping
Data-Flow Functional Computer (DFFC)	LIMSI-CNRS, France	512 Custom built Field Programmable Operator Arrays (FPOA), each with 2 configurable data path; 3-D 8 $\times 8 \times 8$ array of FPGA	Real-time image processing
DTM-1	MITRE Corp., Virginia	16 DTM chips	Each DTM chip is an array of 64 $\times 64$ expandable gate cells.
Enable++	Universitaet Mannheim, Germany	16 Xilinx 4013; 11 Xilinx 5005; 12 MB RAM	Modular FPGA multiprocessor system; Supports Systolic Parallel C (SPC)
Functional Memory Computer	University of Hawaii	8 Xilinx 4010; 3 Xilinx 4013; 1 MB SRAM	Memory-mapped FPGA-based interconnect
GANGLION	IBM Research Division, San Jose	24 Xilinx 3090; 24K PROM	Neural Network
Marc-1	University of Toronto, Toronto	25 Xilinx 4005; 6 MB RAM	$256K \times 64$ instruction memory; $256K \times 32$ data memory with Weitek 3364 math processor
MORRPH-ISA	Virginia Polytechnic institute, Blacksburg	6 Xilinx chips (any combination of 4008, 4010, 4013, 4025)	2×3 mesh network; Real-time image processing

Table 3.2: A summary of custom computing machines. (Contd.)

Product name	Designer	Typical system configuration	Comments
Perle-1	DEC, Paris Research Lab, France	24 Xilinx 3090	Fixed mesh; RSA Encryption
PRISM-II	Brown University, Providence	3 Xilinx 4010 per processing node; 20 nodes connected by a reconfigurable interconnection topology;	Each PRISM-II board is a node in the ARMSTRONG III parallel processor
ProBoard	NTT Optical Network Systems Lab. Japan	12 PROTEUS FPGA (SRAM-based, custom designed)	FPGA-based interconnect switches
Reconfigurable Neural Network Server	Norwegian Institute of Technology, Norway	16 Xilinx FPGAs; 16 TMS320c30 available;	Neural networks
Rapid Prototyping engine for Multiprocessors (RPM)	University of Southern California, LA	7 Xilinx 4013s per board	Used for multiprocessing experiments
Spectrum	Gigaops, Berkeley	32 Xilinx 4013s	Video computing, DSP using a C-like HDL
Splash 2	IDA Supercomputing Research Center, Bowie	16 Xilinx 4010s per board	Variety of applications including DNA pattern matching and image processing
TERAMAC	Hewlett-Packard, CA	108 custom FPGAs, 32 MB SRAM	Rapid turn around of designs to allow investigation of alternate computing ideas
Virtual Computer	Virtual Computer Corporation, CA	52 Xilinx 4013	General purpose computing
Zelig	University of York, UK	32 Xilinx 3090, Ring topology	Cellular automaton, image morphology, rank filtering, Neural Networks

Table 3.2: A summary of custom computing machines.

Splash 2 is one of the leading FPGA-based custom computing machine designed and developed by the Supercomputing Research Center [32]. An interesting observation is the trend in the languages used for algorithm development on these platforms. With smaller systems, the trend has been to use schematic design entry. In most other systems, VHDL or Verilog has been used. Recently, efforts are being made to support subsets of C and C++ [70]. For SIMD mode of operation, dbC is being developed for Splash 2. Note that, after the design entry stage, the other stages need to use vendor dependent tool sets.

3.4 Splash 2

The Splash 2 system consists of an array of Xilinx 4010 FPGAs, improving on the design of the Splash 1 which was based on Xilinx 3090s [10]. Figure 3.3 shows a system-level view of the Splash 2 architecture. Splash 2 is connected to the host through an interface board that extends the address and data buses. The Sun host can read/write to memories and memory-mapped control registers of Splash 2 via these buses. The major components of the Splash 2 system are described below. Each Splash 2 processing board has 16 Xilinx 4010s as PEs ($X_1 - X_{16}$) in addition to a seventeenth Xilinx 4010 (X_0) which controls the data flow into the processor board. Each PE has 512 KB of memory. The Sun host can read/write this memory. The PEs are connected through a crossbar that is programmed by X_0. There is a 36-bit linear data path (SIMD Bus) running through all the PEs. The PEs can read data from their respective memory. A broadcast path also exists by suitably programming

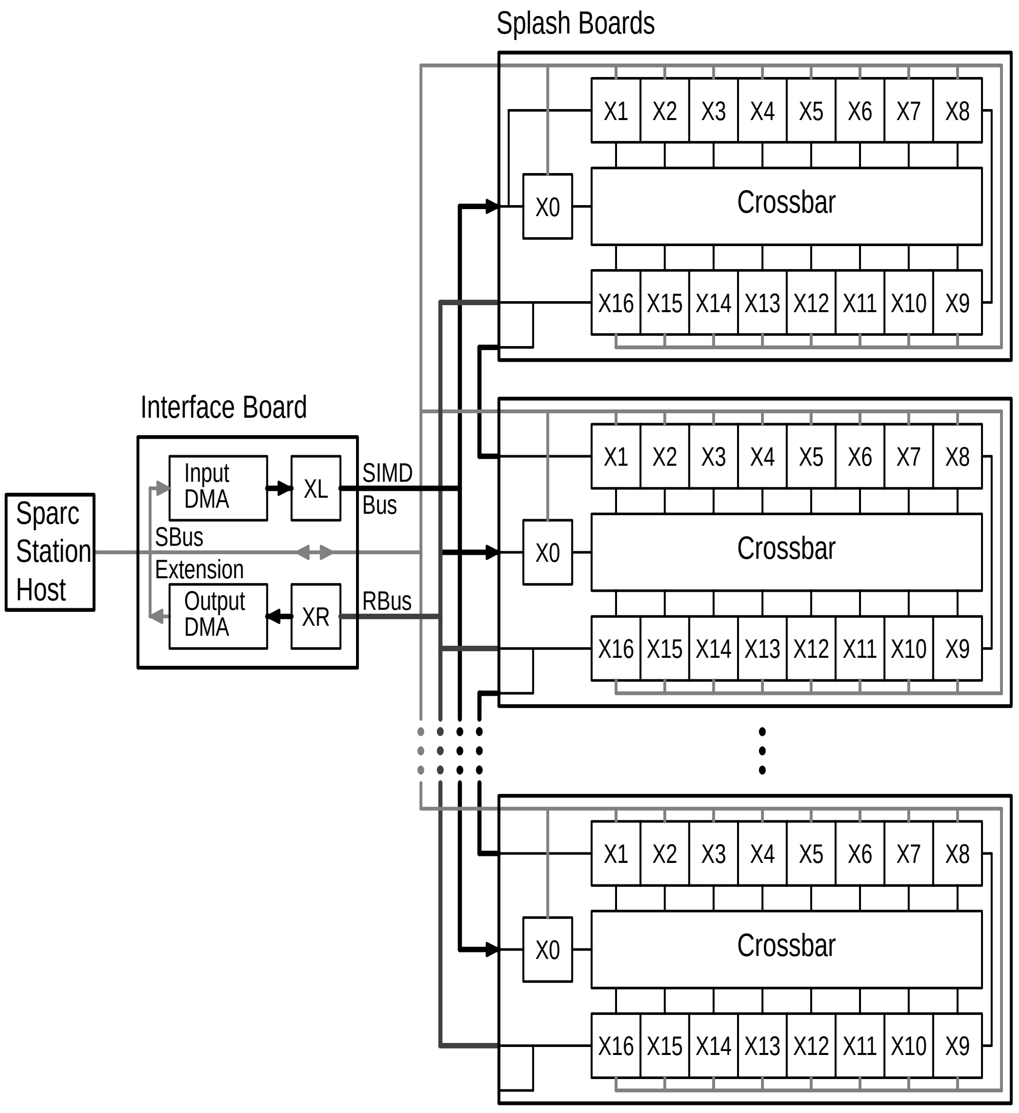


Figure 3.3: Splash 2 architecture.

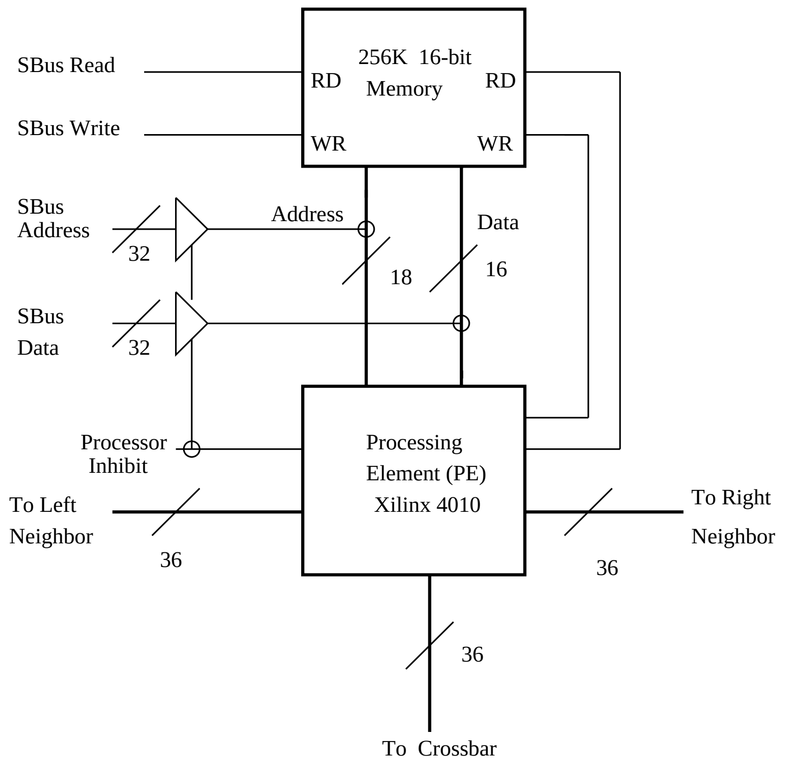


Figure 3.4: A Processing Element (PE) in Splash 2.

X_0. The processor organization for a PE is shown in Figure 3.4. The Splash 2 system supports several models of computation, including PEs executing single instruction on multiple data (SIMD mode) and PEs executing multiple instructions on multiple data (MIMD mode). It can also execute the same or different instructions on single data by receiving data through the global broadcast bus. The most common mode of operation is systolic in which the SIMD bus is used for data transfer. Individual memory available with each PE makes it convenient to store temporary results and tables. One of the important aspects of Splash 2 is the support for symbolic hardware debugging using the symbolic debugger, called $t2$. The symbolic hardware debugger supports many useful facilities for stepping through hardware, observing bus signal values and other functions to aid development process.

3.5 Programming paradigm

Programming an FPGA-based computer is different from usual high-level programming in C or Fortran. The design automation process consists of two steps: simulation and synthesis. The programming flow is shown in Figure 3.5. In simulation, the logic which is designed using VHDL is verified. This involves comparing the results of the VHDL simulation with those obtained manually or by a sequential program. In synthesis, the main concern is to achieve the best placement of the logic in an FPGA in order to minimize the timing delay. At this point in the design process, the logic circuit may or may not fit on a single FPGA (i.e., being able to map it to the configurable logic blocks (CLBs) and flip-flops which are available internal to an FPGA). If the logic does not fit, then the designer needs to revise the logic in the VHDL code and the process is repeated. Once the logic is mapped to CLBs, the timing for the entire digital logic is obtained. In case this timing is not acceptable, the design process is repeated.

To program a Splash 2, we need to program each of the PEs (X_1- X_{16}), the crossbar, and the host interface. The crossbar sets the communication paths between PEs. In case the crossbar is used, X_0 needs to be programmed. The host interface takes care of data transfers in and out of the Splash 2 board. A special library is available for these facilities for VHDL programming as described in [11].

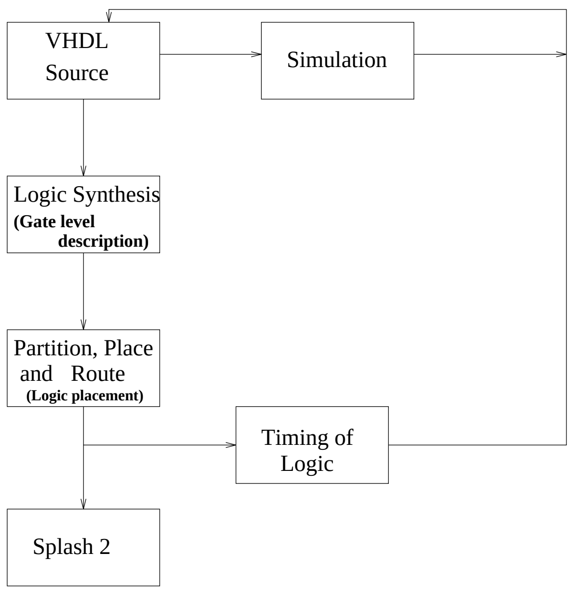


Figure 3.5: Programming flow for Splash 2.

3.6 Logic synthesis

The synthesis process involves the following stages: (i) VHDL to net-list translation: to obtain a vendor specific net-list from the VHDL source code; (ii) partition, placement and routing: to fit the logic generated onto a physical PE; (iii) net-list to bit stream translation; and (iv) bit stream to raw file generation. As a result of the placement, delay analysis can be carried out using the vendor model of the devices. The partition, placement and routing stage is the most complex phase of the synthesis stage. Often, this needs to be repeated by changing the initial random seed to get a better placement.

In the final stage, the software components of the algorithm can be integrated with the hardware. The host-interface carries out the following stages: (i) loading a raw file onto each PE, (ii) configuring the crossbar usage, (iii) initializing PE memory if required, (iv) transfer data, and (v) reading the result. The host uses a set of routines callable by a C program.

Currently, efforts are being made to provide a C-like language, called dbC [77], to program Splash 2 to keep the hardware architecture and communication issues transparent to the end users. The programming model supported in dbC is that of a SIMD processor array with a host processor controlling instruction sequencing for the PEs.

3.7 Software environment of Splash 2

A programmer of Splash 2 needs to be aware of the features of the software environment of Splash 2. The four phases involved in a design process are: (i) simulation, (ii) synthesis, (iii) debugging and (iv) execution from host. The steps involved are shown in Figure 3.6.

The facilities available for the four stages are described below.

- Simulation: Currently, the user has to specify the designs in VHDL. Splash has two entities declared for the X_0 and other X_is. The entity for X_0 is named 'Xilinx_control_part' and the entity for the other X_is is named 'Xilinx_processing_part'. These entities have a predefined port declaration corresponding to the various buses and interface signal they handle. A description and explanation of the entities is available in [32]. There are several library packages available to the programmer. The 'Splash2' package has the definitions of constants, types and functions. A set of padding routines are provided in the 'components' package. The third package consists of Xilinx defined hard-

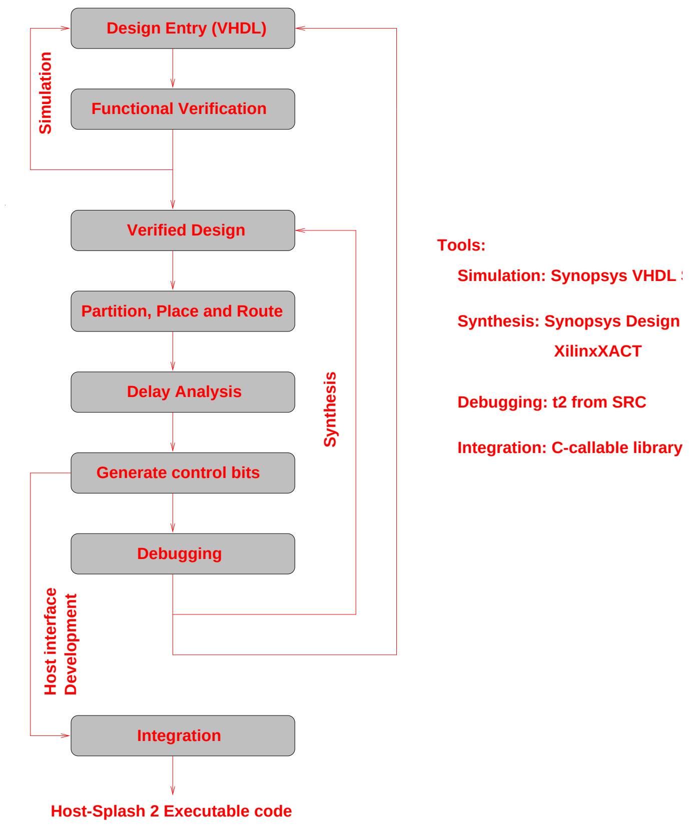


Figure 3.6: Steps in software development on Splash 2.

macros. In later version of Xilinx software, these hardmacro definitions are not required during synthesis. The root module for simulation is called 'system'. It instantiates two models: (i) S2boards and (ii) Interface corresponding to the processor board and the interface board in the Splash 2 system respectively. In a predefined VHDL template for these two, the user is expected to change the architecture names for the two entities 'xilinx_control_part' and 'xilinx_processing_part'. The simulator can handle more than one processing boards by properly setting the value of the generic 'Number_Of_Boards'.

- Synthesis: The synthesis process uses the 'Design Compiler' and 'FPGA Compiler' from Synopsys to translate the VHDL code to Xilinx Netlist Format

(XNF) files. This process is carried out by a Unix shell script 'vhdl2xnf'. The next stage of synthesis is to place and route the logic and generate the control bit stream for the Xilinx 4010-based PEs and also obtain an estimate of the processing speed of the processing speed using 'xdelay' utility. This process is carried out by yet another shell script 'xnf2bit'. In the event the logic could be placed and routed, an estimate of the delays can be analyzed using 'xdelay'.

- Debugging: One of the important phases in development is debugging the bit stream generated on the actual hardware. For this purpose, a hardware symbolic debugger called 't2' is used. The inputs to the '*t*2' is a 'raw' file obtained by associating a bit file for each of the PEs and a definition for the crossbar. Using 't2', the logic can be single stepped, so that intermediate values can be examined on the hardware. A Unix shell-like user interface is provided.

- Execution on host: After the 'raw' file has been ascertained to work on hardware, a C host program interface needs to be written. For this purpose, a set of C-callable routines are available. A run-time library needs to be linked with the code. It should be noted that for the program to run, a device driver for Splash hardware should be resident in the system memory.

3.8 Case study: Image segmentation on Splash 2

The overall process of system design using Splash 2 is described in this section using an algorithm for page layout segmentation. For ease of presentation, a simple image segmentation algorithm has been chosen to illustrate the various design stages and effort involved in a mapping exercise involving Splash 2. A more robust image segmentation technique is described in chapter 5.

3.8.1 Sequential algorithm

The sequential algorithm for page layout segmentation is based on the mean and the variance of gray values at a pixel in a 7×7 window (neighborhood). A pixel will be assigned one of the three labels representing background, text and halftone. The page segmentation algorithm has three stages of computation, namely, (i) mean gray value in a window, (ii) variance of the gray value in a window, and (iii) final label assignment. The flow chart of the segmentation algorithm is shown in Figure 3.7. The input to the algorithm is the gray level scanned image of the document and the output is the labeled image, where each pixel is assigned one of the three class labels. A sample input image and the segmentation result produced by this algorithm are shown in Figure 3.8. This page segmentation algorithm takes about 90 seconds of CPU time on a SPARCstation 20 for a $1{,}024 \times 1{,}024$ image. The output of the segmentation algorithm (shown in Figure 3.8(b)) is fed into a postprocessing stage [107] to place boxes around regions of interest. This "block" representation of Figure 3.8(a) is shown in Figure 3.8(c).

There are two main phases in the segmentation algorithm, not considering the post processing stage. The computation of mean and variance is done in a parallel fashion. Mean value can be computed by convolving the input image with a 7×7

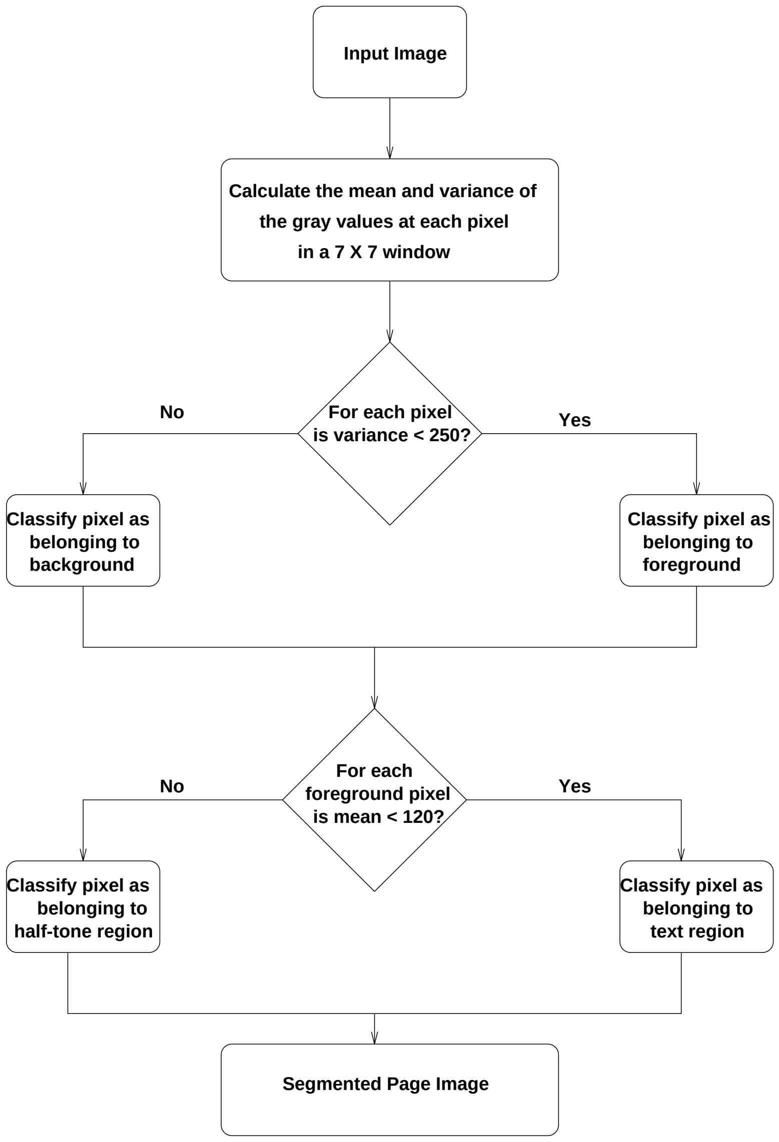

Input Image
Calculate the mean and variance of the gray values at each pixel in a 7 X 7 window
For each pixel is variance < 250?
No
Yes
Classify pixel as belonging to background
Classify pixel as belonging to foreground
For each foreground pixel is mean < 120?
No
Yes
Classify pixel as belonging to half-tone region
Classify pixel as belonging to text region
Segmented Page Image

Figure 3.7: Flow chart of a simple page segmentation algorithm.

mask with all 1's. The expression for the variance σ^2, in a $n \times n$ window can be written as follows:

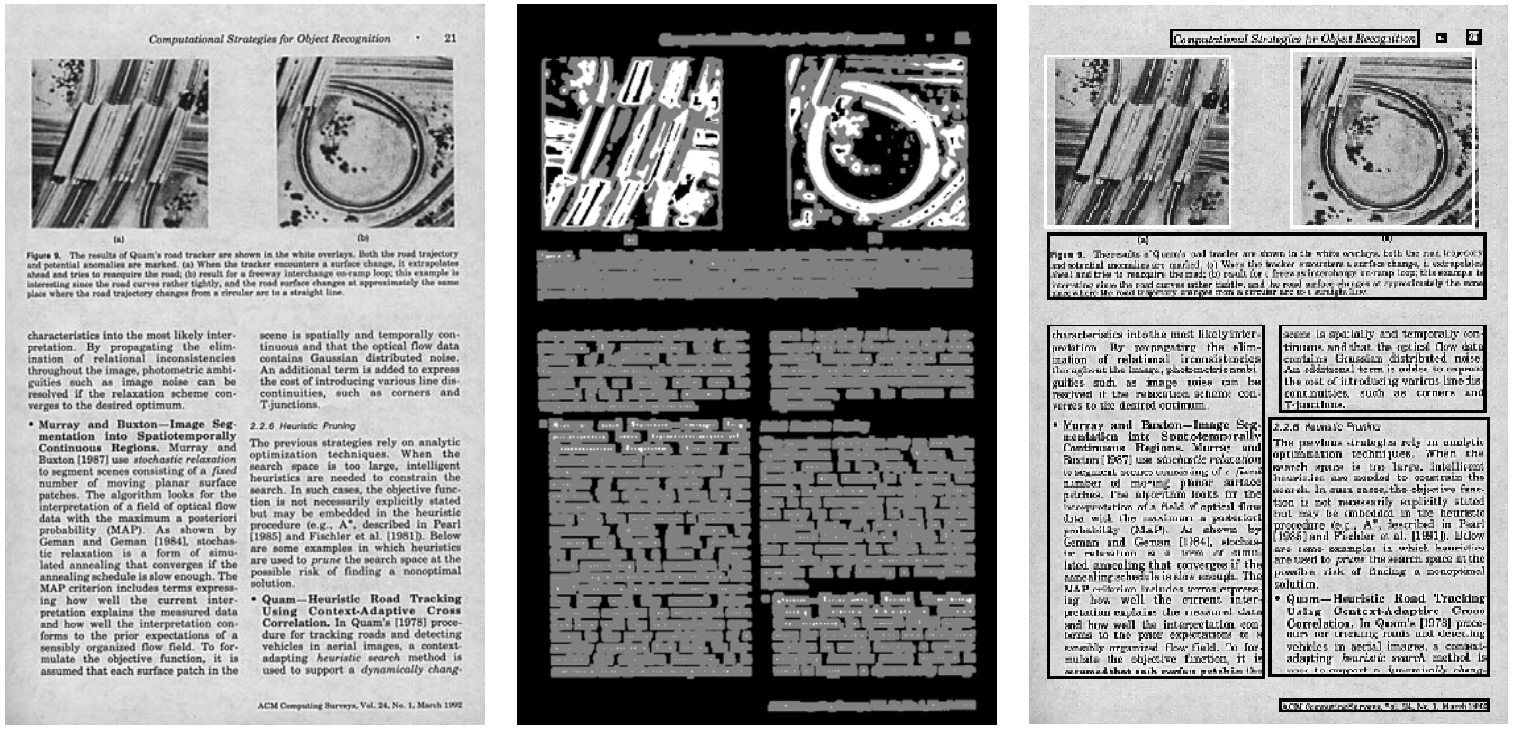
$$\sigma^2 = \frac{(\sum_i \sum_j I_{ij}^2) - N\mu^2}{N}, \tag{3.1}$$

where I_{ij} is the gray value at pixel (i, j), $i, j = 1, \ldots, n$, μ is the mean of the gray value in the $n \times n$ window and $N = n^2$ is the total number of pixels in the window. From Eq. (3.1), it can be easily seen that the overall computation can be split into two stages. In the first stage, the sum of squares of the gray values can be computed. In the second stage, the variance can be computed which needs the mean value, μ, as an input. The sum of squares is easily translated to a convolution operation again with a mask of all 1's. The convolution algorithm will be explained in later chapters in more detail. At this stage it is sufficient to assume that a Splash 2 implementation for convolution is available. Subimages of 32×32 pixels are processed at a time with a boundary of 3 pixels on all the sides of the subimage.

3.8.2 Mapping of image segmentation algorithm on Splash 2

As described earlier, the stages in the design process are (i) VHDL code simulation; (ii) synthesis of the bit stream files for the PEs being used; and (iii) host interface development. The VHDL code for the PEs is included in Appendix A. The VHDL programs are first used with the Splash simulator. The 'makefile' for the simulator is also included in the appendix. During simulation, the signals described in the VHDL programs can be traced. A sample simulation waveform for the algorithm is shown in Figure 3.9. The signals of the VHDL design for X_{15} are traced with respect to the system clock. X_{15} receives the mean from X_7 and the variance from X_{14}. The final label assigned to a pixel is the traced value of the signal 'data'.

The next phase is to synthesize the control bit streams from the VHDL code. For this purpose, 'Synopsys' and 'Xilinx' CAD tools are used.



Computational Strategies for Object Recognition • 21

(a) (b)

Figure 9. The results of Quam's road tracker are shown in the white overlays. Both the road trajectory and potential anomalies are marked. (a) When the tracker encounters a surface change, it extrapolates ahead and tries to reacquire the road; (b) result for a freeway interchange on-ramp loop; this example is interesting since the road curves rather tightly, and the road surface changes at approximately the same place where the road trajectory changes from a circular arc to a straight line.

characteristics into the most likely interpretation. By propagating the elimination of relational inconsistencies throughout the image, photometric ambiguities such as image noise can be resolved if the relaxation scheme converges to the desired optimum.

- **Murray and Buxton—Image Segmentation into Spatiotemporally Continuous Regions.** Murray and Buxton [1987] use *stochastic relaxation* to segment scenes consisting of a *fixed* number of moving planar surface patches. The algorithm looks for the interpretation of a field of optical flow data with the maximum a posteriori probability (MAP). As shown by Geman and Geman [1984], stochastic relaxation is a form of simulated annealing that converges if the annealing schedule is slow enough. The MAP criterion includes terms expressing how well the current interpretation explains the measured data and how well the interpretation conforms to the prior expectations of a sensibly organized flow field. To formulate the objective function, it is assumed that each surface patch in the scene is spatially and temporally continuous and that the optical flow data contains Gaussian distributed noise. An additional term is added to express the cost of introducing various line discontinuities, such as corners and T-junctions.

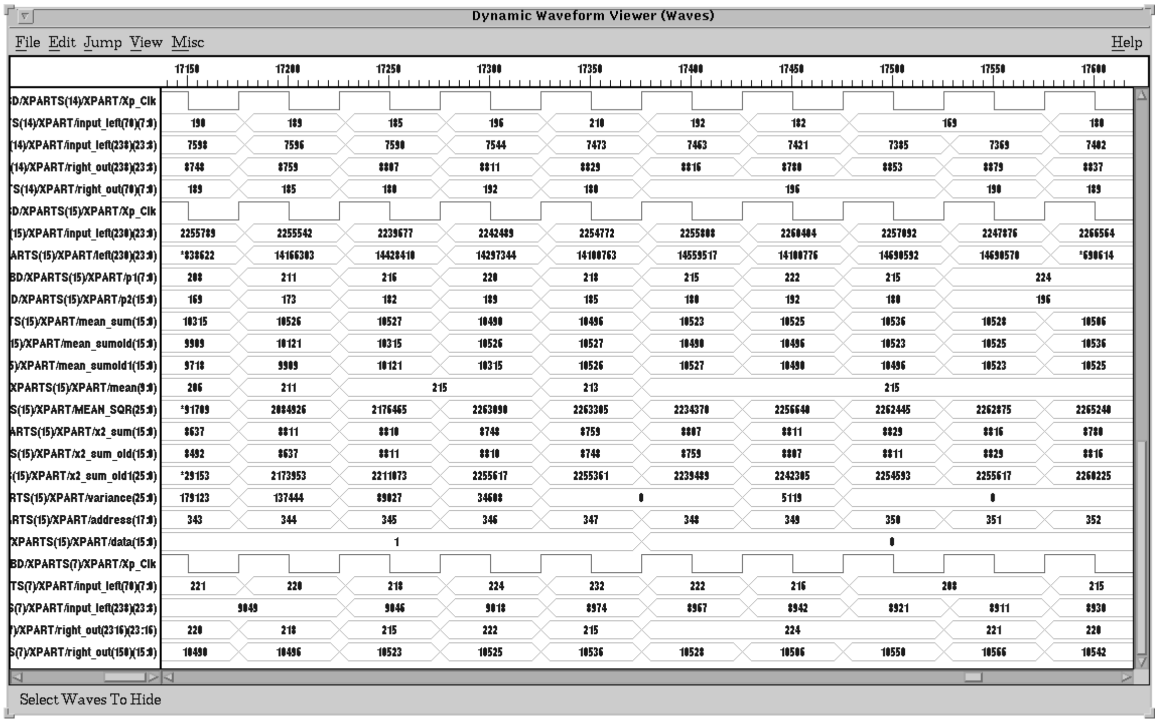
2.2.6 Heuristic Pruning

The previous strategies rely on analytic optimization techniques. When the search space is too large, intelligent heuristics are needed to constrain the search. In such cases, the objective function is not necessarily explicitly stated but may be embedded in the heuristic procedure (e.g., A*, described in Pearl [1985] and Fischler et al. [1981]). Below are some examples in which heuristics are used to *prune* the search space at the possible risk of finding a nonoptimal solution.

- **Quam—Heuristic Road Tracking Using Context-Adaptive Cross Correlation.** In Quam's [1978] procedure for tracking roads and detecting vehicles in aerial images, a context-adapting *heuristic search* method is used to support a *dynamically chang-*

ACM Computing Surveys, Vol. 24, No. 1, March 1992

Figure 3.8: Page Layout Segmentation. (a) Input gray-level image; (b) Result of the segmentation algorithm; (c) Result after postprocessing.



```
                                  17150    17200    17250    17300    17350    17400    17450    17500    17550    17600
D/XPARTS(14)/XPART/Xp_Clk
S(14)/XPART/input_left(70)(7:0)   190      189      185      196      210      192      182      169               180
(14)/XPART/input_left(230)(23:0)  7598     7596     7590     7544     7473     7463     7421     7385     7369     7402
(14)/XPART/right_out(230)(23:0)   8748     8759     8807     8811     8829     8816     8780     8853     8879     8837
S(14)/XPART/right_out(70)(7:0)    189      185      180      192      180      196                        190      189
D/XPARTS(15)/XPART/Xp_Clk
(15)/XPART/input_left(230)(23:0)  2255789  2255542  2239677  2242489  2254772  2255808  2260404  2257092  2247876  2266564
ARTS(15)/XPART/left(230)(23:0)    *838622  14166303 14428410 14297344 14100763 14559517 14100776 14690592 14690570 *690614
BD/XPARTS(15)/XPART/p1(7:0)       208      211      216      220      218      215      222      215      224
D/XPARTS(15)/XPART/p2(15:0)       169      173      182      189      185      180      192      180      196
TS(15)/XPART/mean_sum(15:0)       10315    10526    10527    10490    10496    10523    10525    10536    10528    10506
15)/XPART/mean_sumold(15:0)       9909     10121    10315    10526    10527    10490    10496    10523    10525    10536
5)/XPART/mean_sumold1(15:0)       9718     9909     10121    10315    10526    10527    10490    10496    10523    10525
XPARTS(15)/XPART/mean(9:0)        206      211      215               213      215
S(15)/XPART/MEAN_SQR(25:0)        *91709   2084926  2176465  2263090  2263305  2234370  2256640  2262445  2262875  2265240
ARTS(15)/XPART/x2_sum(15:0)       8637     8811     8810     8748     8759     8807     8811     8829     8816     8780
S(15)/XPART/x2_sum_old(15:0)      8492     8637     8811     8810     8748     8759     8807     8811     8829     8816
(15)/XPART/x2_sum_old1(25:0)      *29153   2173953  2211073  2255617  2255361  2239489  2242305  2254593  2255617  2260225
RTS(15)/XPART/variance(25:0)      179123   137444   89027    34608    0                 5119     0
RTS(15)/XPART/address(17:0)       343      344      345      346      347      348      349      350      351      352
XPARTS(15)/XPART/data(15:0)       1                                   0
BD/XPARTS(7)/XPART/Xp_Clk
TS(7)/XPART/input_left(70)(7:0)   221      220      218      224      232      222      216      208               215
S(7)/XPART/input_left(230)(23:0)  9049              9046     9018     8974     8967     8942     8921     8911     8930
)/XPART/right_out(2316)(23:16)    220      218      215      222      215      224                        221      220
S(7)/XPART/right_out(150)(15:0)   10490    10496    10523    10525    10536    10528    10506    10550    10566    10542
```

Figure 3.9: Simulation results of page layout segmentation on Splash 2.

The 'ppr' utility produces the summary report which contains the CLB occupancy results. A sample summary report is also enclosed in the Appendix A. The timing analysis utility gives an estimate of the peak speed as shown in Figure 3.12. The graph shows the histogram of the number of nets that can run at a given clock speed. Hence, the lowest speed net determines the overall speed of the overall circuit. The lowest speed in Figure 3.12 is 17.1 MHz which is also provided as a text output by the 'timing' utility.

The control bit stream files are first used with the help of the on-line debugger $t2$. Using $t2$, a 'raw file' is created. This raw file can be run on the hardware interactively through $t2$, or through a C program. A sample $t2$ session to generate the 'raw' file from the bit streams is included in the Appendix A. The host interface program written in C reads the input image, creates windows of (subimages) 32 $\times$ 32 pixels

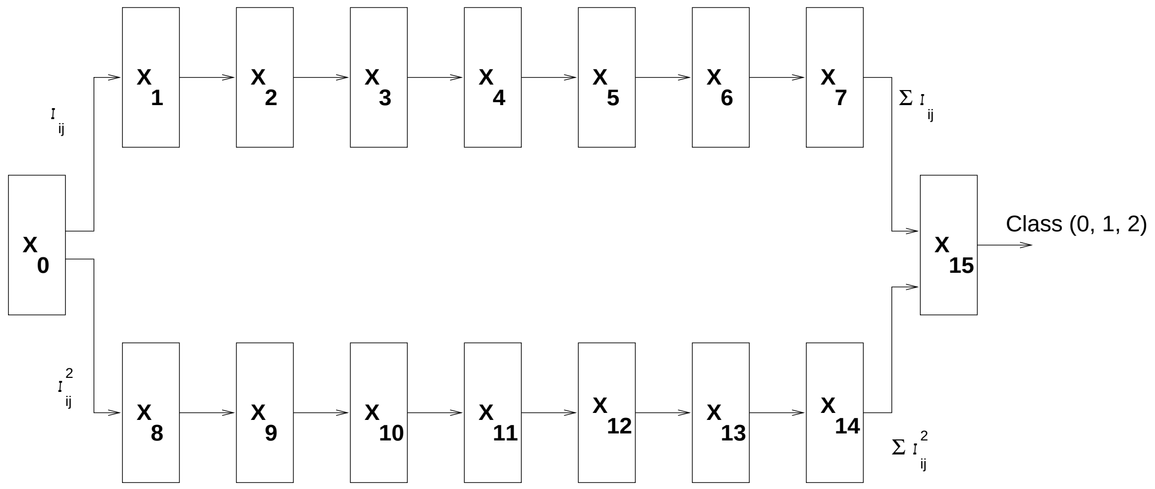


Figure 3.10: Schematic for page layout segmentation on Splash 2.

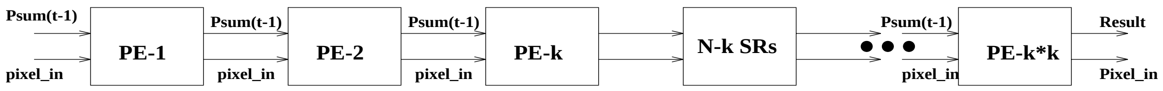


Figure 3.11: Schematic for filtering on Splash 2.

and loads them on X_0 memory. The window size of 32×32 pixels is a restriction laid down by the convolution stage. By running the specified number of clock ticks on Splash, the result is produced and stored in X_{15} memory. The host reads back this result from Splash board. This procedure is repeated for all the possible windows in the input image. The boundaries of the 32×32 windows are shown in Figure 3.13(a). The C code for the host interface and the 'makefile' are included in the Appendix A.

3.8.3 Experimental results

The sequential algorithm has been written in C and tested on several input images. On a SPARCstation 20 (33 MFLOPS, 82 MIPS), it takes about 90 seconds of CPU time to segment a 1,024 $\times$ 1,024 image. The algorithm has been mapped onto Splash

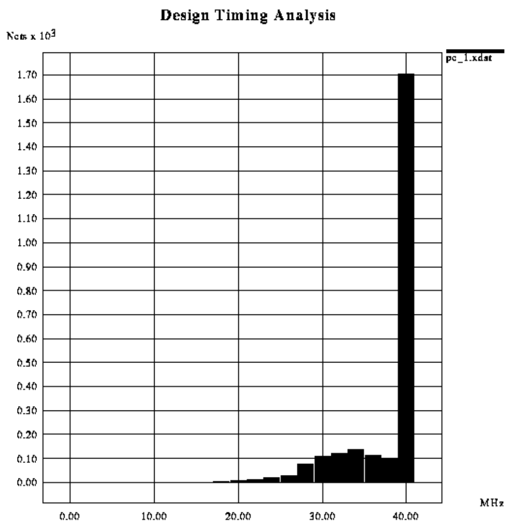


Figure 3.12: Projected speed for X_1.

2. Running the Splash 2 at a clock rate of 5 MHz, the execution time for the page segmentation algorithm is expected to be 360 milliseconds. Therefore, a speedup of close to 250 has been achieved. The simulation results are shown for the PEs X_7, X_{14} and X_{15} in Figure 3.9. The output of X_7 shows the sum of the pixels and the output of X_{14} shows the sum of squares of pixels. Various signals in X_{15} show the computations of mean and variance and the final label assignment. The final labels are stored in the local memory of X_{15} which is read by the host at the end.

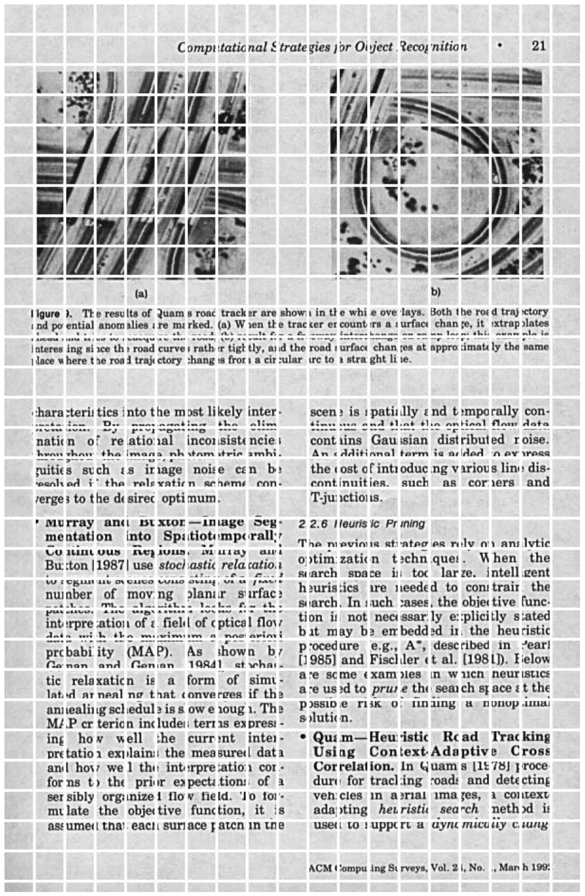
Figure 3.13(b) shows the segmented output of the input image (Figure 3.8(a)) generated by the Splash system. This segmented output is identical to the output generated by the sequential algorithm (see Figure 3.8(b)).

3.9 Summary

We have introduced FPGAs as compute elements. Many custom computing machines have been reviewed. Splash 2 is the platform for the work reported in this thesis. The architecture of Splash 2 has been described. A simple algorithm has been chosen as a case study for demonstrating the software development cycle for Splash 2.

There are many advantages of using FPGAs. The most important aspect is the rapid turn-around time for development and fast prototyping. The user gets the optimal cost/performance ratio compared to designing an ASIC. The financial risks are also minimal as the hardware development uses only off-the-shelf components without involving costly design and fabrication of masks. The number of ICs used for the glue-logic design can be reduced drastically, resulting in a more reliable system.

A complex FPGA can consist of millions of gates. Hence, manually placing and routing the logic is not feasible. This results in limiting the performance of the system to the capabilities of the CAD tools used. For supporting reprogrammability, we pay the price in terms of slower speed compared to an ASIC. The CAD tools are also costly.



Computational Strategies for Object Recognition • 21

(a) b)

Figure 9. The results of Quam's road tracker are shown in the white overlays. Both the road trajectory and potential anomalies are marked. (a) When the tracker encounters a surface change, it extrapolates [illegible] interesting since the road curves rather tightly, and the road surface changes at approximately the same place where the road trajectory changes from a circular arc to a straight line.

characteristics into the most likely interpretation. By propagating the elimination of relational inconsistencies throughout the image, photometric ambiguities such as image noise can be resolved if the relaxation scheme converges to the desired optimum.

• Murray and Buxton—Image Segmentation into Spatiotemporally Continuous Regions. Murray and Buxton [1987] use *stochastic relaxation* to segment scenes consisting of a *fixed* number of moving planar surface patches. The algorithm looks for the interpretation of a field of optical flow data with the maximum a posteriori probability (MAP). As shown by Geman and Geman [1984], stochastic relaxation is a form of simulated annealing that converges if the annealing schedule is slow enough. The MAP criterion includes terms expressing how well the current interpretation explains the measured data and how well the interpretation conforms to the prior expectations of a sensibly organized flow field. To formulate the objective function, it is assumed that each surface patch in the scene is spatially and temporally continuous and that the optical flow data contains Gaussian distributed noise. An additional term is added to express the cost of introducing various line discontinuities, such as corners and T-junctions.

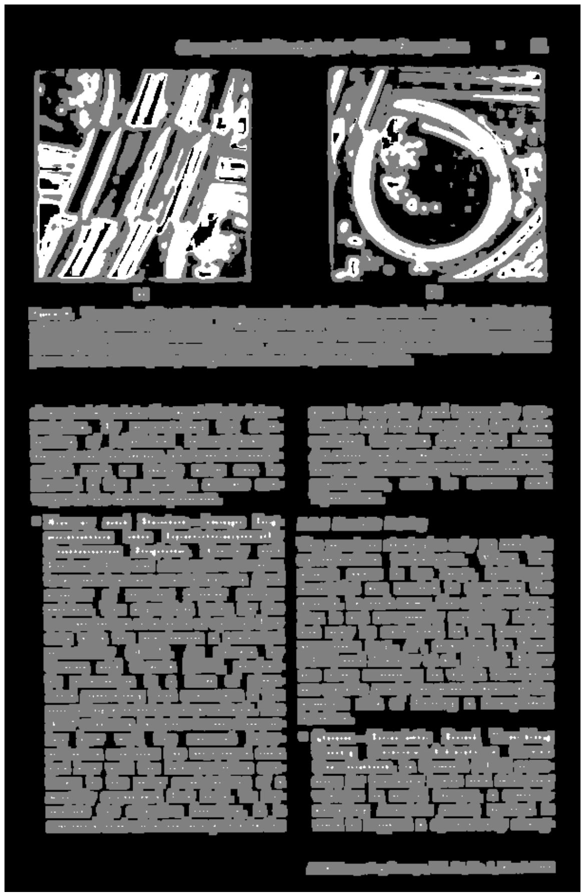
2.2.6 Heuristic Pruning

The previous strategies rely on analytic optimization techniques. When the search space is too large, intelligent heuristics are needed to constrain the search. In such cases, the objective function is not necessarily explicitly stated but may be embedded in the heuristic procedure (e.g., A*, described in Pearl [1985] and Fischler et al. [1981]). Below are some examples in which heuristics are used to *prune* the search space at the possible risk of finding a nonoptimal solution.

• **Quam—Heuristic Road Tracking Using Context-Adaptive Cross Correlation.** In Quam's [1978] procedure for tracking roads and detecting vehicles in aerial images, a context adapting *heuristic search* method is used to support a *dynamically chang*

ACM Computing Surveys, Vol. 2[illegible], No. [illegible], March 199[illegible]

(a)



(b)

Figure 3.13: Page Layout Segmentation using Splash 2. (a) Input gray-level image with 32×32 pixel windows shown; (b) Result of the segmentation algorithm running on Splash.

Chapter 4

Image Convolution

Vision algorithms can be classified into one of the three levels of vision tasks they handle, namely, (i) low-level, (ii) intermediate-level, and (iii) high-level. The image enhancement related activities are usually handled by the low-level operators. In general, this class of algorithms can also be viewed as performing filtering operations. The filtering algorithms can be (i) linear or (ii) non-linear. The point operation-based filters are usually linear and region or neighborhood-based filters are non-linear. The main distinction being that the linear operators are invertible, hence, the operation can be undone by applying the inverse operator. A generalized convolution that can be applied to template matching and morphological filtering of images is described in the next section. Application of convolution and image morphology in fingerprint feature extraction and document image processing is explained.

4.1 Generalized convolution

Convolution is an important operator in digital signal and image processing. Many machine vision systems use 2-dimensional convolution for image filtering, edge detection, and template matching. Generalized convolution of two signals G and W is often expressed as $C = G * W$, where C is the result of applying a convolution mask W to the input signal G. For 2-dimensional discrete images, generalized convolution can be defined [64] as follows:

$$C(i,j) = \psi\phi[f(G(i+s, j+t), W(s,t))] \tag{4.1}$$

where ψ, ϕ and f are three operators.

Table 4.1 shows the values the operators take for many standard operations. In this

task	ψ	ϕ	$f(a,b)$
Correlation	$\sum_s$	$\sum_t$	a * b
Binary erosion	$\cap_s$	$\cap_t$	a & b | $\overline{b}$
Binary dilation	$\cup_s$	$\cup_t$	a & b
Grayscale erosion	Min_s	Min_t	b - a
Grayscale dilation	Max_s	Max_t	a + b

Table 4.1: Generalized 2-D Convolution. ('*': multiplication, '-': subtraction, '+': addition,'&': logical 'AND' operator; '$\overline{b}$: complement of b': logical complement; '|': logical 'OR' operator)

chapter, template matching and morphological operators will be discussed further for mapping onto Splash 2.

4.2 Template matching

Convolution of the 2-D discrete signals f and g can be written as

$$h(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=\infty}^{\infty} f(x-i, y-j) g(i,j). \tag{4.2}$$

For images of finite size (say $N \times N$), and masks of finite size (say $k \times k$), 2-D convolution can be expressed as

$$h(x,y) = \sum_{i=-k/2}^{k/2} \sum_{j=-k/2}^{k/2} f(x+i, y+j) g(i,j). \tag{4.3}$$

The operation described in Eq. (4.3) is also known as template matching, or correlation. The image size and mask size need not, in general, contain a square number of elements.

Typical use of convolution in image processing is for edge detection (e.g., Sobel and Prewitt masks), computing texture coarseness, object location using template matching, and image smoothing using Gaussian masks. The general convolution operator described in Eq. (4.3) includes integer-valued and real-valued input images and masks. Examples of real-valued masks include Gaussian smoothing, and computing texture features using Gabor filters. Most commonly used masks such as Laplacian, Sobel, and Prewitt have not only integer mask values, but the mask values can also be expressed as powers of two. In a typical application involving many image processing stages, each stage should be capable of processing real-valued images. Based on this observation, a taxonomy of convolution operations is defined as shown in Figure 4.1.

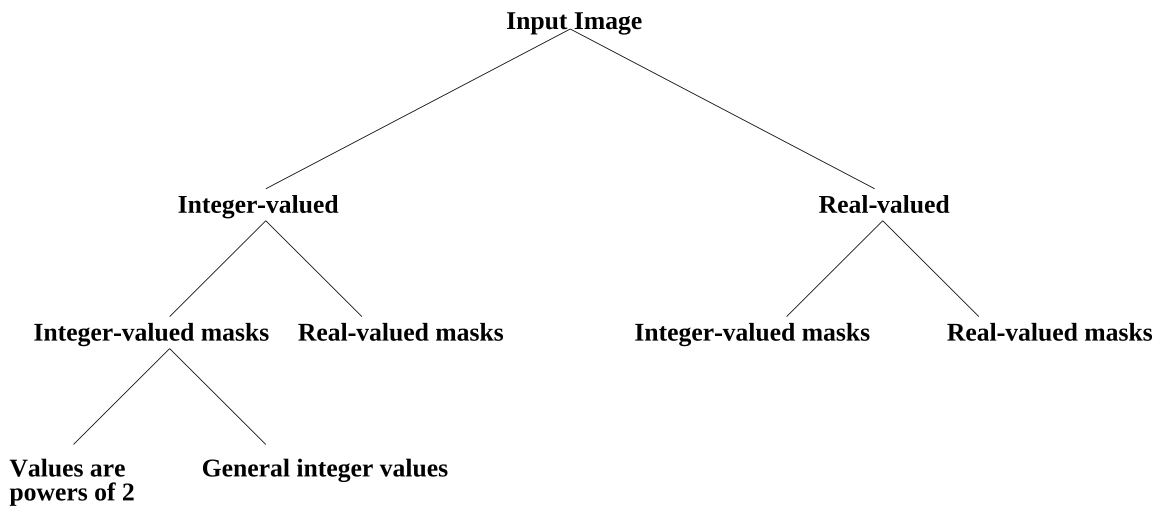


Figure 4.1: A taxonomy of convolution operators.

Convolution on special-purpose architectures has been a topic of substantial interest [63, 130, 181]. A very simple distributed approach for convolution is to split the input image into a set of smaller, possibly overlapping, subimages; the number of subimages is the same as the number of processing elements (PEs). Each PE produces the result for the sub-image it receives. The spatial support for the convolution mask is provided through the overlap at the boundaries or data replication. However, this simplistic approach may not be suitable for all target architectures. For example, if the image is being acquired line by line then this approach may not be efficient as we have to wait till we acquire the whole image. In order to understand the advantages and disadvantages of various algorithms reported in the literature, we need to look at the communication and computation pattern in 2-dimensional convolution. The basic convolution operation is shown schematically in Figure 4.2. Suppose the value of the convolution operation is desired at point (x, y). The center of the mask is placed at

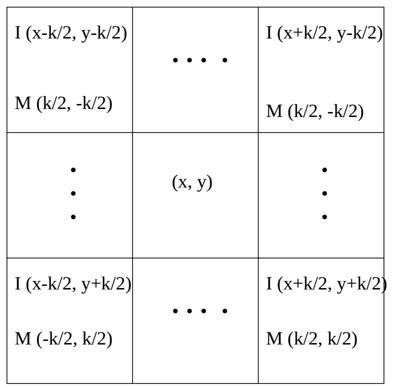


Figure 4.2: The convolution operator. I(i,j) is the input gray image value at pixel (i,j) and M(u,v) is the mask value at (u,v).

(x, y). A point-wise inner product of the image pixels and mask values is computed, followed by a reduction sum operation. This computes the output value at (x, y). The reduction operation can also be a prefix sum, although the intermediate results are not directly useful. This basic set of operations is repeated at all possible (x, y) locations.

The sequential version of the convolution algorithm is very simple and is shown in Figure 4.3. There are four loops in the algorithm and its overall complexity is $O(N^2k^2)$. The simple data partitioning approach described above can reduce the total computation time by a factor equal to the number of available processors, ignoring the extra computations needed in the overlapping areas by each PE.

The previous work described in the literature can be summarized based on the target architecture on which the convolution operation is implemented.

- Systolic: One of the most widely used convolution algorithm is the systolic algorithm by Kung *et al.* [130]. The algorithm is fairly straight forward and also scalable to higher dimensions using the 1-D convolution algorithm as the

```
for i = 1 to N do
    for j = 1 to N do
        sum = 0
        for u = -k/2 to k/2 do
            for v= -k/2 to k/2 do
                sum = sum+I[i+u][j+v]*M[u][v]
        O[i][j] = sum
    end
end.
```

Figure 4.3: Sequential algorithm for 2-dimensional convolution. I(i,j) and O(i,j) are the input and output values, respectively, at pixel (i,j) and M(u,v) is the mask value at (u,v).

building block. In his landmark paper "Why systolic architectures?" [129], Kung described many convolution algorithms on systolic structures. Based on a general inner product computation, Kulkarni and Yen [127] proposed a systolic algorithm for 1-D and 2-D convolutions.

- Hypercube: Fang *et al.* [63] have described an $O(k^2/p^2 + klog(N/p) + logN * logp)$ algorithm, where $1 \leq p \leq k$, using N^2k^2 PEs and an $O(N^2M^2/L^2)$ algorithm using L^2 PEs. Using N^2 PEs, Prasanna Kumar and Krishnan [128] proposed an algorithm with the best time complexity of $O(N^2/K^2 + logN)$. With a fixed number of PEs, the time complexity changes to $O(k^2logk + logN)$.

- Mesh: Many researchers [134, 181] have proposed schemes for convolution on mesh connected architectures. Lee *et al.* [134] use computation along a Hamiltonian path ending at the center of the convolution mask, called the convolution path. Ranka and Shahni [181] do not broadcast the data values, thereby improving the performance of their algorithm by an order of magnitude.

- Pyramid: Pyramid architectures are useful in dealing with multi-resolution images. An $O(k^2 + logN)$ time complexity algorithm is described by Chang *et al.* [35].

- VLSI/ASIC: A number of proposed convolution algorithms are suitable for a VLSI implementation. Chakrabarti and Jaja [33] use a linear array of processors in their algorithm. In [127] convolution is viewed as a generalized inner product and a VLSI implementation for 2-dimensional convolution is described. Ranganathan and Venugopal [180] have described a VLSI architecture for template matching using k^2 PEs and they achieve a time complexity of $O(N^2/2 + K^2)$.

4.3 Image morphology

Mathematical morphology is a powerful tool for image analysis. By proper selection of a structuring element, a number of commonly used algorithms for segmentation, shape analysis and texture can be implemented efficiently. Mathematical morphology has been successfully used in automated industrial inspection, nonlinear filtering, image compression, and biomedical image processing [149, 92].

Mathematical morphology is based on set theory. An image is considered as a discrete set of pixels. For binary images, the set of black pixels is the image set. Gray-level images are treated in a similar way where each pixel has three dimensions, the x- and y-coordinates and the gray value. A *structuring element* is a special user-specified set. The size and shape of the structuring element totally depends on the application. The two basic operators of morphology are defined as follows.

Let A be the image set and S be the structuring element. We define the two basic morphological operators, dilation and erosion, as follows.

- Dilation, denoted by the operator $\oplus$, is defined as follows

  1. a Binary image:

  $$A \oplus S = \{x \mid x \;=\; (a+b)\ for\ some\ a \ \epsilon\ A\ and\ b\ \epsilon\ S\}. \qquad (4.4)$$

  2. Gray level image:

  $$C(i,j) = \max_{u,v \epsilon [0,M-1]} \{A(i+u, j+v) + S(u,v)\}. \qquad (4.5)$$

  where C is the result of gray level dilation and the size of the image is M $\times$ M.

- Erosion, denoted by the operator $\ominus$, is defined as follows:

  1. Binary image:

  $$A \ominus S = \{x \mid (x \;+\; b)\ \epsilon\ A\ for\ every\ b\ \epsilon\ S\}. \qquad (4.6)$$

2. Gray level image

$$C(i,j) = \min_{u,v \epsilon [0,M-1]} \{S(u,v) - A(i+u, j+v)\}. \tag{4.7}$$

Alternatively, the two operators can also be defined using a generalized convolution operator as described in Table 4.1.

- Dilation for binary images:

$$C(i,j) = \bigcup_{u=0}^{M-1} \bigcup_{v=0}^{M-1} A(i+u, j+v) \& S(u,v). \tag{4.8}$$

- Dilation for gray scale images:

$$C(i,j) = \max_{u,v \epsilon [0,M-1]} \{A(i+u, j+v) + S(u,v)\}. \tag{4.9}$$

- Erosion for binary images:

$$C(i,j) = \bigcap_{u=0}^{M-1} \bigcap_{v=0}^{M-1} A(i+u, j+v) \& S(u,v)) | \overline{S(u,v)}. \tag{4.10}$$

- Erosion for gray scale images:

$$C(i,j) = \min_{u,v \epsilon [0,M-1]} \{S(u,v) - A(i+u, j+v)\}. \tag{4.11}$$

The convolution-based definitions are easier to implement in parallel form. Many architectures have been proposed for speeding up the morphological operators. In [27], a method is presented to represent a binary image in a bit-mapped form such that the CPU can handle more pixels at a time. Boomgaard *et al.* [27] also use a decomposition scheme to express a large structuring element as a combination of smaller structuring elements. Crabtree *et al.* [116] redefine 4-connected and 8-connected erosion and dilation operators and claim that their implementation is fast even on personal computers. A logic gate and multiplexer based implementation is described in [143] for gray level images. Lenders *et al.* [138] have proposed a 1-bit systolic processor for binary morphology.

4.4 Application of generalized convolution

In this section, two applications of generalized convolution are described. For computations of dominant ridge directions in a fingerprint, the orientation field [182] model is adopted. In order to remove 'spiky' growths in a fingerprint skeleton image, a morphological filter has been described in [184]. For removing repetitive background in document images, Liang *et al.* [142] proposed a morphological approach. These three applications will be used to demonstrate the application of generalized 2-D convolution.

4.4.1 Orientation field computation using 2-D convolution

The orientation field is used to compute the optimal dominant ridge direction in each 16×16 window or block of the input image. Following steps are involved in the computation of the orientation field for each window.

1. Compute the gradient of the smoothed block. Let $G_x(i,j)$ and $G_y(i,j)$ be the gradient magnitude in x and y directions, respectively, at pixel (i,j) obtained using 3×3 Sobel masks.

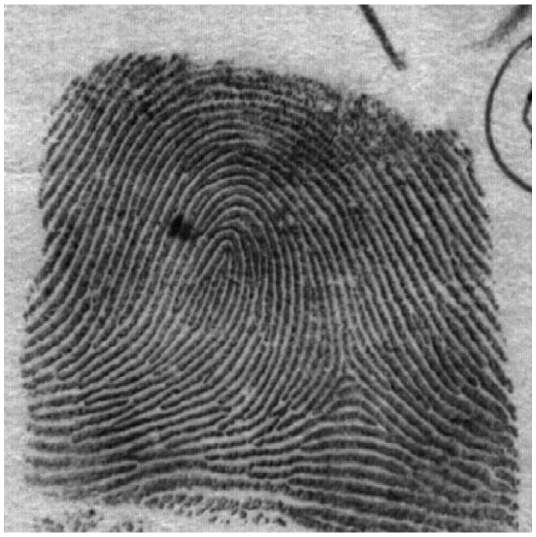
2. Obtain the dominant direction in a 16×16 block using the following equation:

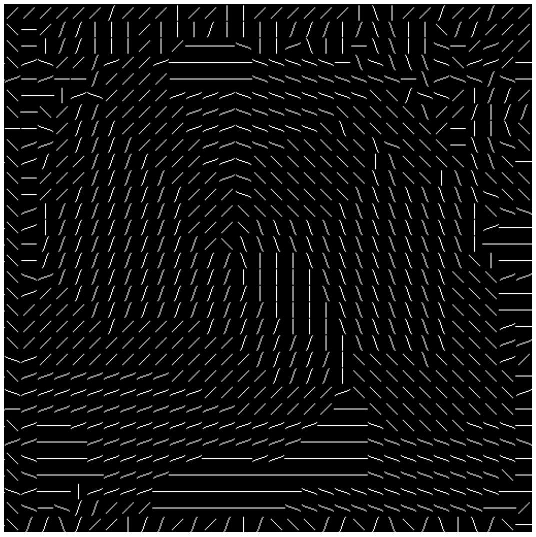
$$\theta_d = \frac{1}{2} tan^{-1} \left(\frac{\sum_{i=1}^{16} \sum_{j=1}^{16} 2G_x(i,j)G_y(i,j)}{\sum_{i=1}^{16} \sum_{j=1}^{16} (G_x(i,j)^2 - G_y(i,j)^2)} \right), G_x \neq 0 \; and \; G_y \neq 0 \qquad (4.12)$$

Note that if either G_x or G_y is zero then the estimate of the dominant direction is trivial (0^o or 90^o). The angle θ_d is quantized into 16 directions. The orientation field obtained using this method is shown in Figure 4.4. The gradient magnitudes (G_x and G_y) at a pixel are computed using an integer-valued convolution mask. This will be implemented on Splash 2.

4.4.2 Skeleton smoothing using image morphology

The binary ridge image needs further processing before the minutiae features can be extracted. The first step is to thin the ridges so that they are single-pixel wide. A skeletonization method described in [196] and available in the HIPS library [202] is used. Unfortunately, the ridge boundary aberrations have an adverse impact on the skeleton, resulting in "hairy" growths (spikes) which lead to spurious ridge bifurcations and endings. Hence, the skeleton needs to be smoothed before minutiae


(a)


(b)

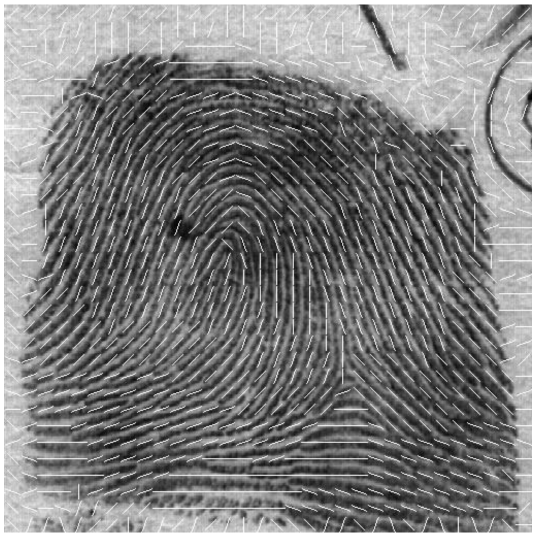

(c)

Figure 4.4: Computation of orientation field; (a) input fingerprint image (512×512); (b) orientation field (for each 16×16 window); (c) orientation field superimposed on the input image.

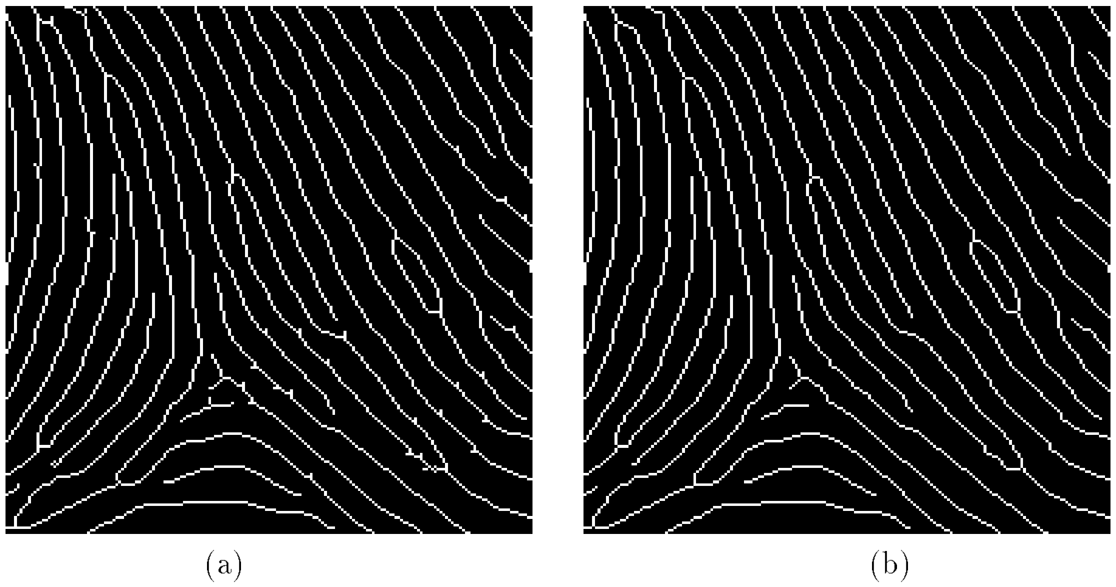


Figure 4.5: Thinned ridges: (a) before spike removal; (b) after spike removal.

points can be extracted. The spikes are eliminated using an adaptive morphological filtering. The filter used is a binary "open" operator with a box-shaped structuring element with all '1's of size 3×3. The structuring element is rotated in the direction orthogonal to the orientation field in the window. The ridge skeletons before spike removal and after spike removal are shown in Figure 4.5. The "open" operator is defined using the two basic dilation and erosion operations.

4.4.3 Background removal in document image processing

Overlapping text and background separation is an important step in document image processing. A morphological filtering-based method is described in [142]. By using an 'open' operator with a suitable size structuring element, repetitive background can be removed as shown in Figure 4.6.

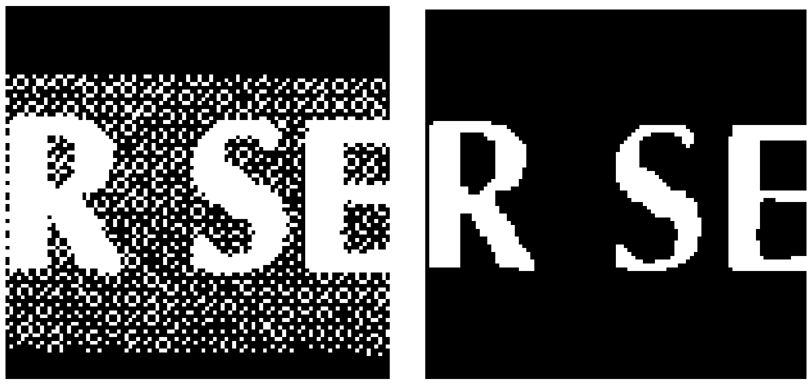


Figure 4.6: Background removal using 'open'. (a) input binary image; (b) output of "open".

4.5 Mapping onto Splash 2

Image processing algorithms, in general, and convolution, in particular, demand high I/O bandwidth. Most of the algorithms implemented on special purpose architectures assume that the data are already available on the PEs. This, in a way, avoids the I/O bandwidth problem of the convolution operation. We do not make this assumption. Jonker [115] argues that linear arrays are better for image processing algorithms. A linear array of PEs operating in a systolic mode offers two advantages: (i) systolic arrays can balance I/O with computations, (ii) the nearest neighbor communication can eliminate the need for a global communication facility for some types of algorithms. One of the preferred modes of computation on Splash 2 is the systolic mode. In this mode, no assumptions are made about the availability of data in the individual PEs. The computations needed in a PE are also fairly simple. This helps us in balancing both the I/O bandwidth and the computation requirements of a PE. Hence, a systolic algorithm for implementing convolution on a Splash 2 is preferred.

1. Assume k PEs are available.
2. The left most PE receives input and the right most PE produces output.
3. $Partial\_sum \leftarrow 0$ for the left most PE.
4. On the i^{th} PE, carry out the following:
    - Receive partial_result and pixel_value from the left neighbor.
    - $partial\_result \leftarrow mask[i] * pixel\_value + left\_result$.
    - Send partial_result and pixel_value to the right neighbor.

Figure 4.7: 1-D systolic algorithm.

First, the simple 1-dimensional convolution algorithm is described. Let us assume that we have k PEs, where k is the size of the $(1 \times k)$ mask. Each PE receives the pixel value and the partial result available so far from its left neighbor. The PE multiplies the pixel value with the mask values assigned to it and adds the partial sum to it. This result and the pixel value are passed to the right neighbor. At the end of the systolic path, we get the convolution result after taking into account the initial latency. The algorithm is shown in Figure 4.7. A schematic diagram of 1-dimensional convolution on a set of PEs connected linearly is shown in Figure 4.8(a).

The above algorithm assumes that the PEs can implement multiplication operation. In a FPGA-based PE, this is not always true. A double precision floating point multiplier needs more logic than what is available in a PE. While we can use the local PE memory to store the multiplication table indexed by the pixel value, this results in an additional delay of one cycle necessary to reference the multiplication result from the memory. This scheme is shown in Figure 4.8(b).

The 2-dimensional convolution is an extension of the 1-dimensional convolution described above. The basic idea is based on the algorithm proposed by Kung *et al.* [130]. The $k \times k$ mask is extended to a $k \times N$ mask with 0's placed at locations where no entry was present. These kN entries are serialized to get a single 1-dimensional mask of kN entries. Now, we can apply the 1-dimensional convolution algorithm outlined above. Note that there are $(N - k)$ locations with 0's as their mask value. Hence, we can simply have $(N - k)$ stages of shift registers. Secondly, for improper positions of this new Nk-element mask we need to ignore some values which are not really part of the output. Finally, we assume that the pixels are being communicated in a raster scan order. Note that we can implement the 1-dimensional convolution algorithm with or without a lookup table. The scheme is shown in Figure 4.8(c).

4.5.1 Implementation issues

The general convolution algorithm needs to be tuned to the special hardware being used. For example, the Splash 2 system has only 16 PEs on a board, and therefore, virtual PEs need to be mapped to physical PEs. The second issue is the number of shift registers which depends on the number of rows in the image. The third issue is the implementation of multipliers needed by the PEs. Following is a summary of our solution to these three issues.

- Large number of mask entries: If mask size (k) is greater than the number of available PEs, then the virtual PEs are mapped to available PEs. In carrying out the mapping, the timing model between PEs must be satisfied.

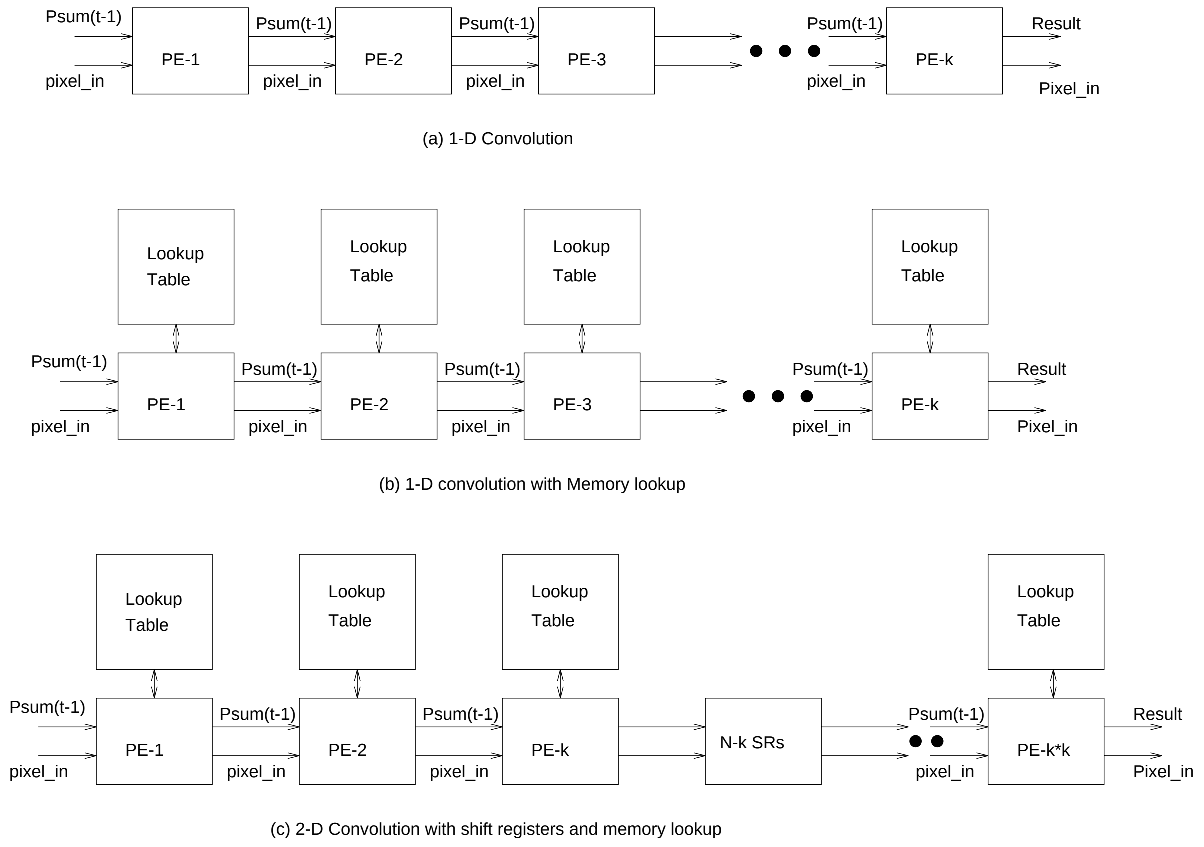

Psum(t-1)
pixel_in
PE-1
PE-2
PE-3
PE-k
Result
Pixel_in
(a) 1-D Convolution
Lookup Table
(b) 1-D convolution with Memory lookup
N-k SRs
PE-k*k
(c) 2-D Convolution with shift registers and memory lookup

Figure 4.8: Systolic schemes for convolution on Splash 2. (a) 1-dimensional convolution; (b) 1-dimensional convolution using memory lookup; (c) 2-dimensional convolution with shift registers and memory lookup.

- Large image width: A large image has to be split into smaller sub-images of some predefined size. In order to handle this, the 2-dimensional convolution has been implemented with a fixed width (32 in our case). Depending on the mask size, the required number of rows and columns at the border are copied.

- Multiplier implementation:

  - Masks with integer values: If mask values are of the type 2^p, where p is an integer then the multiplier in each virtual PE can be replaced with simple bit shifters.

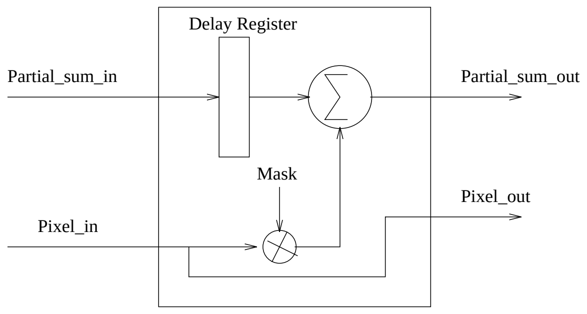


Figure 4.9: A compute element.

- Integer mask values but not necessarily powers of 2. An iterative shift and add algorithm to multiply two eight bit numbers can be implemented.
- Real valued masks: If mask values are normalized [-1, 1], then a suitable scheme for implementation on FPGAs is needed.

The Splash PEs carry out two different activities, namely, (i) additions and multiplications, and (ii) shift operations. We have built following two different type of elements for these activities: (i) compute element, and (ii) shift element. The numbers of compute elements and shift registers are determined by the mask size and image width. The schematics of a compute element is shown in Figure 4.9 and a schematic of the register bank is shown in Figure 4.10.

A brief summary of the analysis of several convolution algorithms is given in Table 4.2 based on the communication facility available on the respective architectures. In a systolic algorithm, the communication overhead is balanced by the computation phase, so no complex communication facility is needed. In this sense, systolic algorithm is better in terms of the total work done and communication simplicity.

We compare our implementation on Splash 2 with implementations on different platforms in terms of the total execution time. The timings for 3×3 convolution-

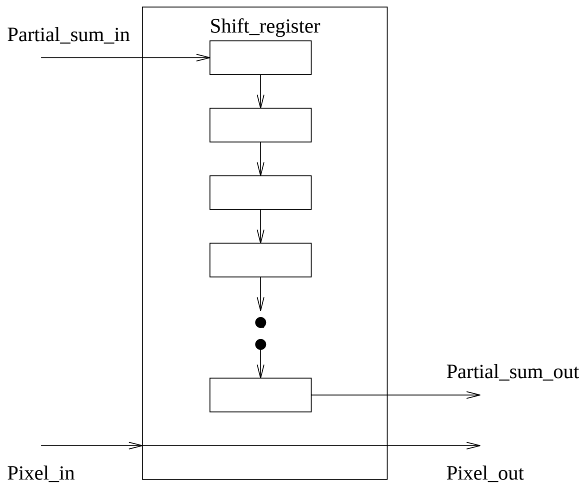


Figure 4.10: A shift register.

Convolution algorithm	Computational Complexity	No. of PEs	Architecture
Ranka & Sahni [181]	$O(k^2 \times$(# of bits in a pixel)	(N^2)	Mesh
Prasanna Kumar & Krishnan [128]	$O(logk)$	$O(N^2k^2)$	Hypercube
Chang *et al.* [35]	$k^2 + logN$	$O(N^2)$	Pyramid
Kung [130] algorithm	$O(N^2)$	$O(k^2)$	Systolic
Ranganathan *et al.* [180]	$O(N^2 + k^2/2)$	$O(k^2)$	VLSI

Table 4.2: Comparative analysis of convolution algorithms; image size is $N \times N$ and mask size is $k \times k$

based Sobel edge detector on a 512×512 image are shown in Table 4.3. The basic sequential convolution algorithm (Figure 4.3) running on different Sun host machines has been timed. In addition, timing on a recently developed i-860 based system from Alacron is reported. The timing results on CM-5 are for edge detection using a set of six 5×5 convolution masks [175].

For implementing the convolution algorithm on Splash 2, three standard edge detectors used in low-level computer vision algorithms have been chosen. These filters have been chosen because the mask values are powers of 2. The filters and their

Machine	Architecture	Time	Remarks
SPARCstation 20 Model 40	Von Neumann	3.60 sec	C-code, timing obtained using 'clock' function
SPARCstation 10 Model 30	Von Neumann	5.85 sec	Same as above
SPARCstation 2	Von Neumann	8.13 sec	Same as above
i-860	Pipelined	51.9 ms	Timings as reported by vendor
Splash 2	FPGA based	13.89 ms	Mask values are powers of 2 (32 pixels base width)
Splash 2	FPGA based	65 ms	Table lookup for multiplication (32 pixels base width)
CM-5	MIMD (512 PEs)	40 ms	Result reported in [175]*
CM-5	MIMD (32 PEs)	605.75 ms	Result reported in [175]*
MasPar-2	SIMD (4K PEs)	84 ms	Result reported in [169]; scaled to 512×512 image
Datacube	Pipelined	13.8 ms	[57]
MVC-150	Pipelined	15 ms	[101]
Data Translation	Spl. hardware	66 msecs	[56]

Table 4.3: Timings for a 3×3 Sobel edge detector for a 512×512 image on different platforms. * Results are for an edge detector based on six 5×5 convolution masks.

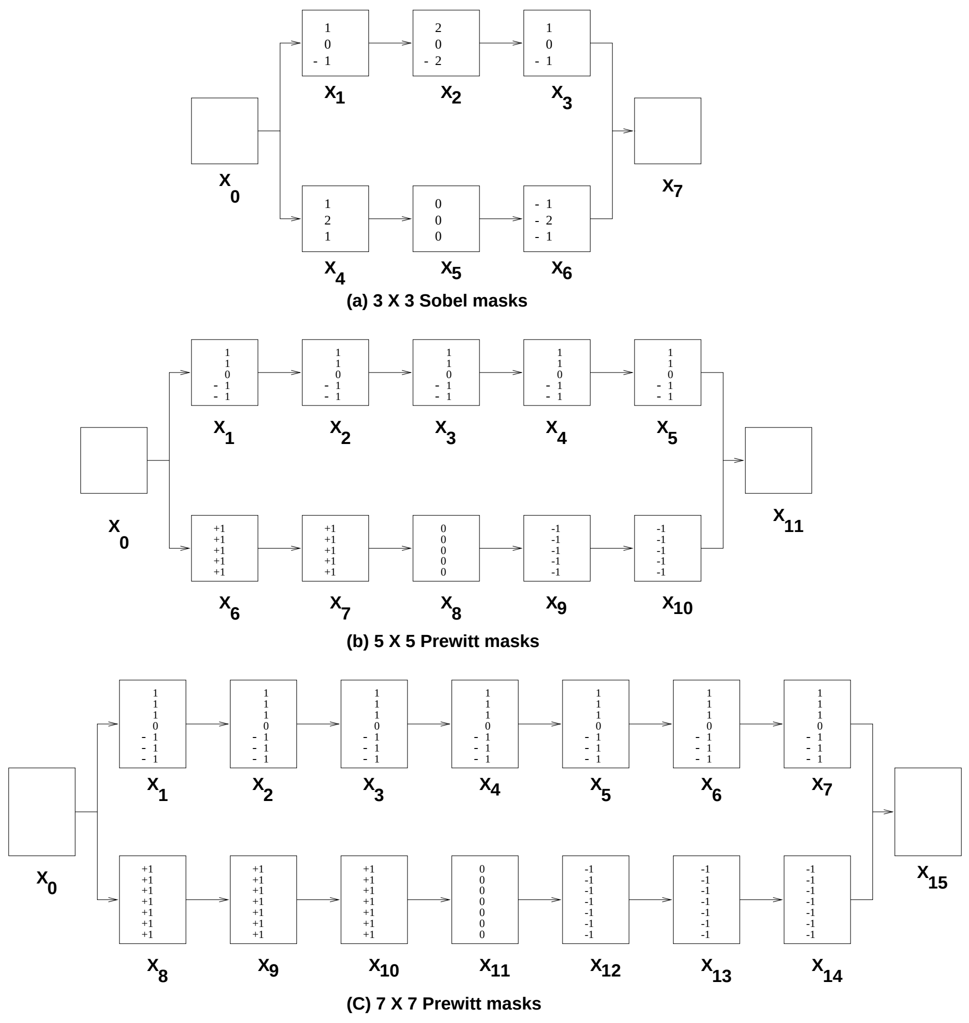


Figure 4.11: Implementation of three edge detection filters on Splash 2. (a) 3 ×3 Sobel masks; (b) 5 × 5 Prewitt masks; (c) 7 × 7 Prewitt masks; X_1 – X_{15} denote the PEs in Splash 2.

	Image Size		
Mask Size	128×128	256×256	512×512
3×3	51.6	206.6	813.3
5×5	54.9	216.6	864.9
7×7	56.6	228.3	914.9

Table 4.4: Run times (in milliseconds) of edge detection on Splash 2 for various image and convolution mask sizes.

mappings on Splash 2 PEs are shown in Figure 4.11. These operators have two convolution stages and a final stage to compute the edge (gradient) magnitude at each pixel by computing the absolute sum of the two convolution output images. Each PE accommodates the required shift registers for that stage. The gradient magnitude outputs for the house image using a 3×3 Sobel edge detector, a 5×5 Prewitt edge detector and a 7×7 Prewitt edge detector are shown in Figure 4.12. The timings for these edge detectors on Splash 2 are shown in Table 4.4.

Our approach for implementing convolution operation on Splash 2 is different in many ways from the approach taken by Peterson *et al.* [172]. The main differences are: (i) we are not limited by a fixed mask size of 8×8 as done in [172]. For smaller masks, Peterson *et al.* [172] have used the same 8×8 masks filled with zeros. This may be due to hard-coded model of computing on Splash 2; (ii) Peterson *et al.* use all the 16 PEs for implementing their algorithm. We use a fewer number of PEs for smaller mask sizes ($k < 8$). Therefore, in our approach, we will have more PEs available for implementing other image processing algorithms; and (iii) our mapping on Splash 2 results in a higher performance of nearly 100 frames per second compared to 15 frames per second reported by Peterson *et al.* [172].

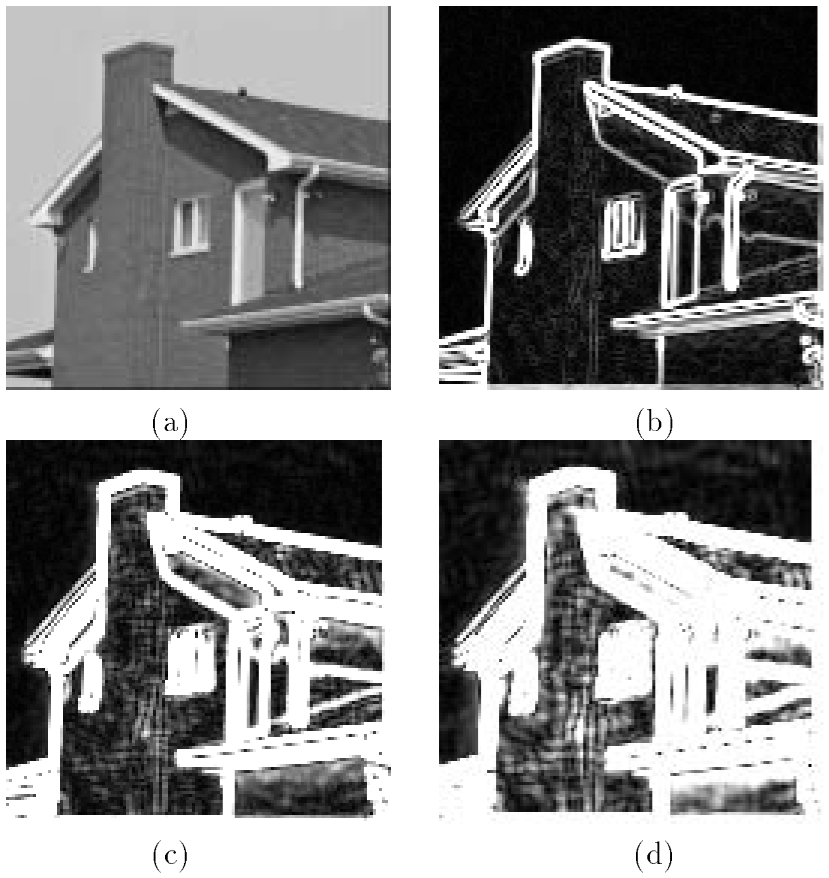


Figure 4.12: Results of 2-dimensional convolution. (a) Input image; (b) 3×3 Sobel edge detector output; (c) 5×5 Prewitt edge detector output; (d) 7×7 Prewitt edge detector output.

4.6 Analysis of convolution on Splash 2

The purpose of this analysis is two fold. First, the total number of clock cycles needed for 2-D convolution on a $M \times M$ image using a $k \times k$ mask is computed. Second, the number of physical FPGAs needed in the operation is estimated. It can be easily seen from the algorithm that both these quantities are deterministic and can be computed using a general formula derived below. The number of FPGAs is being computed to see the effect of number of CLBs in an FPGA on convolution algorithm. Although our target hardware (Xilinx 4010) has fixed number of CLBs, with advances in FPGA technology, more and more CLBs are becoming available in a chip. The convolution algorithm can benefit from a large number of CLBs as it is dependent on the number of adders and shift registers that can be accommodated in a chip.

Following notations and assumptions are used in the analysis:

1. Image size = $M \times M$,
2. Block image size = $N \times M$,
3. Mask size = $k \times k$,
4. Number of CLBs needed to realize a 16-bit adder (hard macro/relatively place macros) = a
5. Number of CLBs needed to realize a 16-bit parallel loadable shift register = r
6. Total number of CLBs in a FPGA chip = c
7. Occupancy factor (% of CLBs occupied in a chip) = f
8. Partial results (sums) are 16-bit wide and the data is 8-bit wide.

As described in the 2-D convolution algorithm, the input image is split into equal sized data segments of width N and height M (original height) as shown in Figure 4.13. The following computations show the number of cycles needed to complete the 2-D convolution on a data segment of size $N \times M$. Note that there is an overlap of $k/2$ pixels on all sides of the data. We define

$$\text{Total time} = \text{Latency} + \text{Computation time} + \text{Flush out time},$$

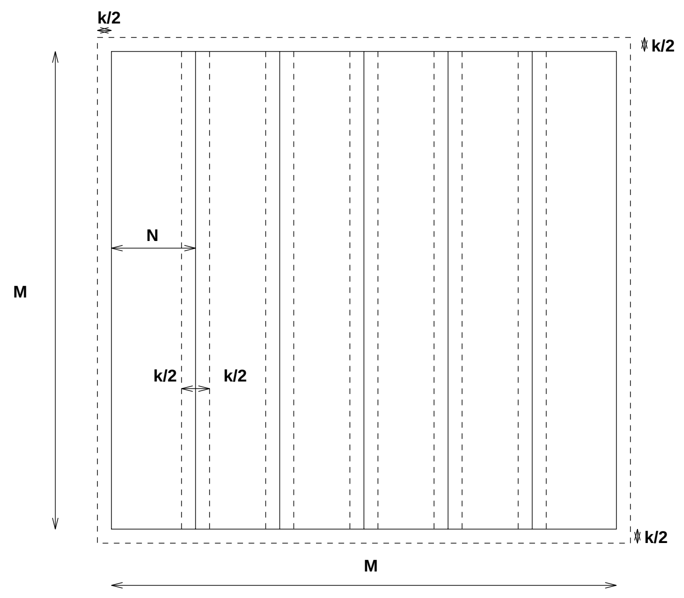


Figure 4.13: Notation used in the analysis of 2-D convolution algorithm.

where

Latency = initial number of clock cycles before the first valid data appears,

computation time = number of clock cycles actually used in the computation, and

Flush out time = number of clock cycles to flush out the left over data elements of the present dataset.

Based on the above definitions,assumptions and notations, the following values are arrived at.

Latency $= N \times k$;

Flush out time $= N \times k$;

Computation time $= (N + k) \times (M + k)$;

Wrap around loss $= (k - 1) \times M$;

No. of data segments $= M/N$;

Using these values, the total number of clock cycles for all the $\frac{M}{N}$ segments can be obtained as follows.

Total number of clock cycles $= \frac{M}{N}(2N \times k + (N+k) \times (M+k))$;

We define efficiency as the ratio of ideal number of clock cycles and actual number of clock cycles taken by the algorithm as shown below.

efficiency $= \frac{((M+k)\times(M+k))}{Total\ number\ of\ cycles}$;

The total number of cycles and the efficiency values for commonly observed values of M, N, and k are shown in Table 4.5. The total number of clock cycles and the efficiency for different assigned values of M, N and k is computed using the definitions described earlier. The maximum efficiency of 0.953 is possible for $M = 1,024$, $N = 64$ and $k = 3$.

The communication complexity of the algorithm is simple due to the choice of the systolic algorithm. The computation and communication steps are overlaid in each clock, hence, no communication overheads are involved.

For estimating the number of FPGAs needed, we note that there are two subcases: (a) masks that are powers of two, and (b) general floating point masks. Let P be the desired number of FPGA chips. Then,

(a) No. of adders needed/virtual PE = 1;

Total number of adders = k;

No. of registers = k;

No. of shift registers = $N - k$;

In order to be able to accommodate the desired number of CLBs, the

M	N	k	$(M+k) \times (M+k)$	No. of cycles	Efficiency
256	16	3	67081	80272	0.836
256	16	5	68121	90256	0.755
256	16	7	69169	100368	0.689
256	16	9	70225	110608	0.635
256	32	3	67081	74056	0.906
256	32	5	68121	79816	0.853
256	32	7	69169	85640	0.808
256	32	9	70225	91528	0.767
256	64	3	67081	70948	0.945
256	64	5	68121	74596	0.913
256	64	7	69169	78276	0.884
256	64	9	70225	81988	0.857
512	16	3	265225	316192	0.839
512	16	5	267289	352544	0.758
512	16	7	269361	389152	0.692
512	16	9	271441	426016	0.637
512	32	3	265225	291472	0.910
512	32	5	267289	311184	0.859
512	32	7	269361	331024	0.814
512	32	9	271441	350992	0.773
512	64	3	265225	279112	0.950
512	64	5	267289	290504	0.920
512	64	7	269361	301960	0.892
512	64	9	271441	313480	0.866
1,024	16	3	1054729	1254976	0.840
1,024	16	5	1058841	1393216	0.760
1,024	16	7	1062961	1531968	0.694
1,024	16	9	1067089	1671232	0.639
1,024	32	3	1054729	1156384	0.912
1,024	32	5	1058841	1228576	0.862
1,024	32	7	1062961	1301024	0.817
1,024	32	9	1067089	1373728	0.777
1,024	64	3	1054729	1107088	0.953
1,024	64	5	1058841	1146256	0.924
1,024	64	7	1062961	1185552	0.897
1,024	64	9	1067089	1224976	0.871

Table 4.5: Number of clock cycles and efficiency for commonly observed values of M, N, and k

following inequality must hold.

$P * f * c \leq k(k * a + N * r);$ .... (i)

(b) No. of adders needed/virtual PE = 8; (8 bit normalized coeff. value)

Total number of adders = $8 * k$;

No. of registers = k;

No. of shift registers = $N - k$;

Again, in order to accommodate the required CLBs, the following inequality must be satisfied.

$P * f * c \leq k(8 * a * k + N * r);$ .... (ii)

For Xilinx 4010 FPGA chip, $c = 400$, $a = 10$ and $r = 9$; for an easy placement of the logic, we assume $f = 0.75$. When $k = 7$, and $N = 64$, we can see that the inequality (i) is satisfied with $P = 7$ in case (a) and the inequality (ii) is satisfied with $P = 14$ in case (b).

4.7 Discussion

A generalized 2-D convolution is of significant importance for low-level vision tasks. Most of the image processing accelerators support template matching and morphological processing as independent functions. Although speeds up to 40 MHz are available for convolution with 4×4 masks using these accelerators, they suffer from following shortcomings. The implementations are restricted to mask sizes already defined by the designer. Secondly, operations which need to be performed on the outputs of convolution stages cannot be performed on the same hardware at the same time. For example, the computation of Sobel edge detection needs a computation of both S_x and S_y (gradient magnitudes in x and y directions) at the same time, followed by an addition of $|S_x|$ and $|S_y|$. Similar constraints exist in computation of the dominant ridge direction in fingerprint images. This was possible using the PEs and the crossbar communication on the processor board of Splash 2. Any operation on the output of convolution has been made possible by the other PEs available on Splash 2 which could be programmed for a separate instruction after receiving the convolution results through the crossbar. For the morphological 'open' operation, the results of the first stage could be fed to the second stage using the SIMD bus. There is no loss of synchronism in connecting the two stages. On the commercial pipelined processors, the performance of the standard accelerators drops to 10 MHz for an 8×8 mask size. The performance on the Splash 2 remains unchanged as we increase the mask size.

4.8 Summary

In this chapter, sequential algorithms for low-level vision tasks, 2-D convolution and image morphology have been introduced. Applications of these basic operators have been shown in realizing stages of minutiae feature extraction in fingerprint images and removing repetitive background in document image processing. The mapping of the generalized convolution onto Splash 2 is described. The analysis of the algorithm mapping has been carried out. Results in terms of synthesis speeds obtained have been compared with several other hardware systems.

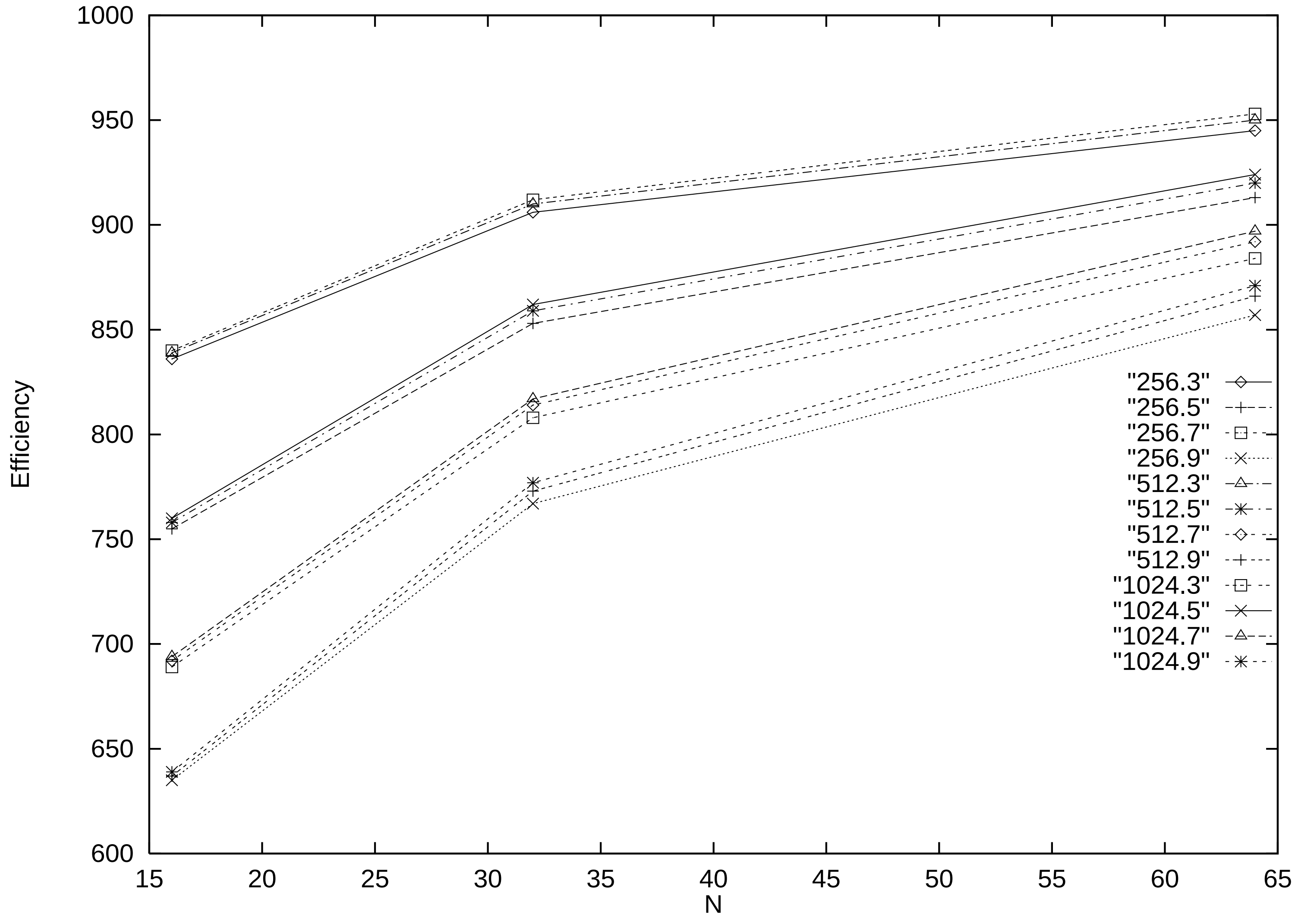


Figure 4.14: Efficiency versus width for various image and mask sizes.

Chapter 5

Image Segmentation

The process of spatial partitioning of an image into mutually exclusive connected image regions is known as image segmentation [90, 168]. Each region is expected to be homogeneous with respect to a defined property. Typically, image segmentation is carried out in the early stages of a vision system to facilitate image representation and interpretation. An image segmentation problem is analogous to the problem of pattern clustering in the sense that we need to define the similarity criterion between pixels or pattern vectors and the number of segments or clusters [105]. Most well known algorithms for image segmentation are based on the following approaches: (i) thresholding or clustering, (ii) boundary detection, and (iii) region growing. The similarity between pixels is based on the notion of homogeneity which involves gray level, color, texture and optical flow information.

5.1 Page layout segmentation

In an automated document image understanding system, page layout segmentation plays an important role for segmenting text, graphics and background areas. Such a segmentation allows us to apply character recognition algorithms to only text regions. A number of algorithms have been reported for page layout segmentation [170]. Haralick *et al.* [89] use image morphology-based techniques. Jain and Bhattacharjee [103] have used a multi-channel filtering approach based on Gabor filters. Jain and Chen [104] have used color information along with Gabor filter outputs for page layout analysis applicable to locating address labels. Recently, Jain and Karu [106] have proposed an algorithm to learn texture discrimination masks needed for segmentation. The performance of this approach for page layout segmentation has been demonstrated by Jain and Zhong [107].

The page segmentation algorithm by Jain and Zhong [107] has three stages of computation, namely, (i) feature extraction, (ii) classification, and, (iii) postprocessing. The feature extraction stage is based on a set of twenty masks obtained by the learning paradigm proposed in [106]. The second stage is a multistage feedforward neural network with 20 input nodes, 20 hidden nodes and three output nodes. The connection weights and other parameters of the neural network have been learned for document images using the approach described in [106]. The 7×7 masks for the feature extraction stage have been shown to be "optimal" in the sense of minimizing the classification error for the three-class (text and line drawings, half-tone and background) segmentation problem. The post-processing stage involves removing small

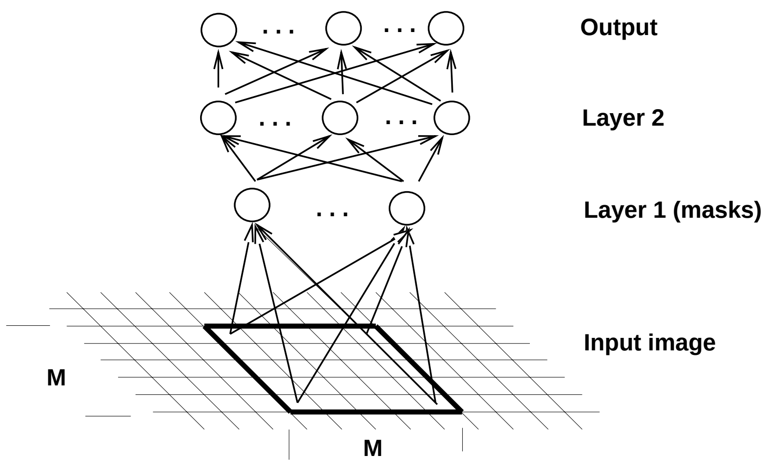
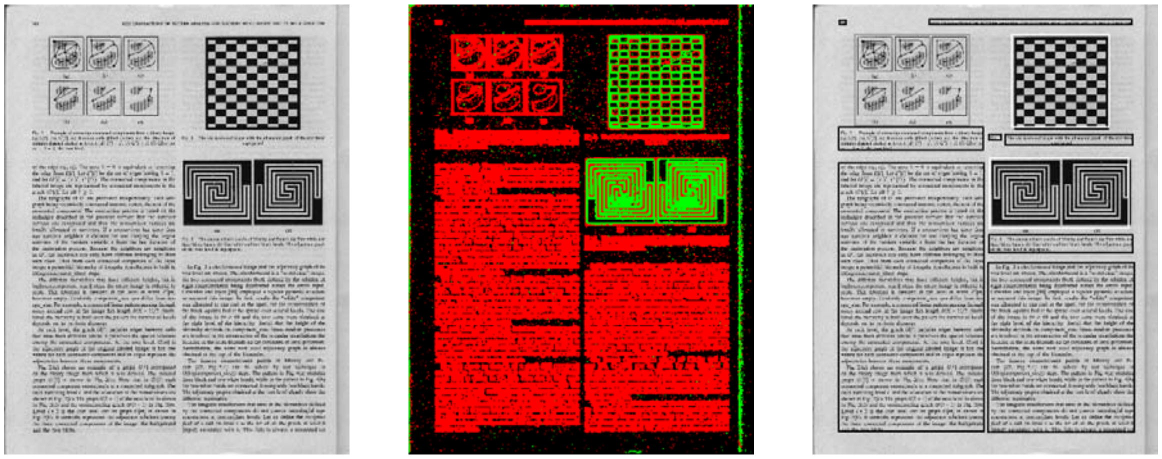


Figure 5.1: Schematic of the page layout algorithm.

noisy regions and placing rectangular blocks around homogeneous identical regions. The schematic diagram of the segmentation algorithm is shown in Figure 5.1 (M = 7 in our implementation). The input to the algorithm is the gray level scanned image of the document and the output is the labeled image, where each pixel is assigned one of the three classes. A sample input image and the segmentation result produced by this algorithm are shown in Figure 5.2. The input gray level is image shown in Figure 5.2(a). The three-level segmentation results obtained by the sequential algorithm is shown in Figure 5.2(b) where the background is shown by black pixels, the text areas are shown in gray pixels and the the the graphics areas are shown in white. Figure 5.2(c) shows the results after postprocessing the segmentation result where the segmented areas are enclosed in rectangular boxes. The text areas are enclosed by black boxes and graphics areas are enclosed by white boxes. The twenty 7×7 masks and the weights of the neural network stage are listed in the Appendix B. This page segmentation algorithm takes about 250 seconds of CPU time on a SPARCstation 20 for a 1,024 $\times$ 1,024 image.



(a) (b) (c)

Figure 5.2: Page layout segmentation. (a) Input gray-level image; (b) Result of the segmentation algorithm; (c) Result after postprocessing.

The computational requirements of this page segmentation algorithm can be summarized as follows. There are twenty 7×7 feature extraction masks. Each feature vector (with 20 elements) needs 20×49 multiplications and 48×20 additions. The size of a typical input document image is $1{,}024 \times 1{,}024$ pixels. Therefore, the filtering stage requires of the order of 10 billion multiplications and 10 billion additions. The neural network classifier requires the following computations: for every 20-dimensional feature vector, we need 400 multiplications in the first stage and 60 multiplications in the second stage. The first stage also involves 20×19 additions and the second stage needs 3×19 additions. Note that the feature values for the second stage are floating point numbers compared to the integer-valued input pixels in the feature extraction phase. For a $1{,}024 \times 1{,}024$ image, there are 460 million floating point multiplications in the classification stage.

5.2 Mapping onto Splash 2

The two main phases in the segmentation algorithm are filtering and multilayer feed-forward network. The filtering stage is carried out first. The Splash 2 system is then reconfigured to implement the neural network classifier. In this section, we will address the mapping issues for these two stages.

An important design parameter that needs to be decided is the data widths at different stages. The input is already known to be 8-bits. The result of the filtering stage has been decided to be 12-bit signed integers which is then fed to the neural network stage. The neural network hidden layer has a 2's complement 12-bit output and, to be consistent, the output layer has the same width. The 2's complement representation is used for the integers in all the stages.

5.2.1 Filtering

The mask values for this application have been derived using the learning paradigm described in [106]. The twenty 7×7 feature extraction masks are real-valued. The filtering algorithm is implemented as a 2-D convolution with floating-point multiplication. The 2-D convolution presented in the previous chapter has been extended to real-valued masks. One of the limitations with the current FPGAs is the minimal support for floating point operations. This disadvantage can be turned into an advantage by developing suitable multiplication operators needed to carry out the floating point arithmetic. As the input is an 8-bit gray valued image, the floating point multiplication is converted to a fixed point multiplication using shift and add iteratively.

1. Normalization: the mask coefficients are normalized between [-1, 1] by dividing the mask values by a constant chosen to be a power of two and greater than or equal to the largest (absolute) mask value.
2. Floating point multiplication: multiplication of an integer by a fraction is considered as successive sums of partial results.
3. Renormalization: the result is scaled back by the normalization constant used.

Figure 5.3: Fixed-point multiplication.

The algorithm is shown in Figure 5.3. With this approach, a general-purpose floating point multiplier has been replaced by a series of adders by taking advantage of the input range being [0, 255]. The overall schematic design for the 2-D convolution is already described in the previous chapter.

The algorithm needs the services of the host to do the normalization of the coefficients. The fractional value is represented as an 8-bit value. Thus, a maximum of eight adders are needed. For each coefficient in the 7×7 mask, the number of adders are different. As the total number of adders to realize a mask rise sharply, more FPGAs are needed. Hence, a full Splash 2 processor board is dedicated for each filter. Each of the twenty masks is programmed separately to get a compact mapping.

5.2.2 Analysis of the filter mapping

To realize twenty filters, a total of twenty processor boards are required to complete the segmentation in one pass. With a 2-board system, it takes 10 passes over the input image. The masks being fixed, this approach is acceptable. The twenty control bit streams for the twenty filters are available to reuse the available processor boards dynamically. Note that the input image can reside on the X_0 host and the number of passes to realize the twenty filters can read the image from X_0 instead of host. Reprogramming the processor board with a different control stream leaves the external memory unchanged.

An on-chip multiplier approach is preferred over a look-up table mode in the implementation to reduce the number of memory accesses to the external memory. Recall that in the 2-D implementation, a full row of mask-values are mapped to a single physical PE. The external memory being single ported, seven accesses to memory would require 7 cycles, thus, scaling down the performance linearly by 7.

5.2.3 Neural network classifier

The neural network used in the classification stage is a multilayer perceptron (MLP). The present approach concentrates on the classification activity and not in the learning of connection weights which is typically done off-line. Hence, it is assumed that the network architecture and the weights have already been determined. The weights are assumed to be real-valued numbers in general. In our application the network has twenty input nodes, each node corresponding to one of the texture filter outputs (real-valued). The intermediate (hidden) layer has 20 nodes and the output layer has 3 nodes. A MLP consists of several perceptrons interconnected in a feedforward manner as shown in Figure 5.4.

Artificial neural networks (ANNs) exhibit six types of parallelism [161]. Most commonly, three types of parallelism namely, layer level, node level and weight parallelism are exploited. Two different design approaches have been taken to map a

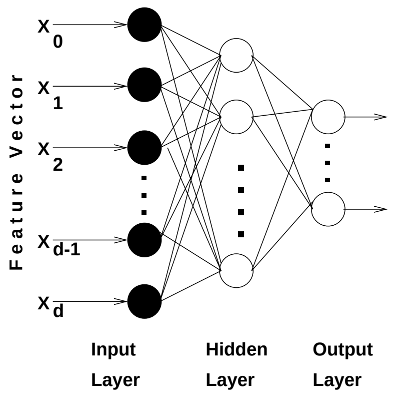


Figure 5.4: A multilayer perceptron.

MLP onto Splash 2. In the first case, the design is modular and is capable of accommodating any number of layers and neurons. In contrast, the second method handles only three layered (input, hidden and output) networks fully utilizing the hardware capabilities. However, both the methods utilize a common building block - a neuron. A single neuron implements a perceptron. A MLP is realized as an interconnection of several neurons.

Many special-purpose implementations of neural networks have been described in the literature. A survey of parallel architectures for neural networks is given in [200]. Mueller and Hammerstrom [157] describe design and implementation of CNAPS – a gate array based implementation of ANNs. A single CNAPS chip consists of 64 processing nodes connected in a SIMD fashion using a broadcast interconnect. Each processor has 4K bytes of local memory, a multiplier and ALU, and dual internal buses. The adder and multiplier can perform signed 8-, 16-, or 32-bit integer arithmetic. Using Xilinx XC 3090 FPGAs, Cox *et al.* describe the implementation of GANGLION [52]. A single board caters to a fixed neural architecture of 12 input nodes, 14 hidden nodes and 4 output nodes. Using the CLBs, 8×8 multipliers have

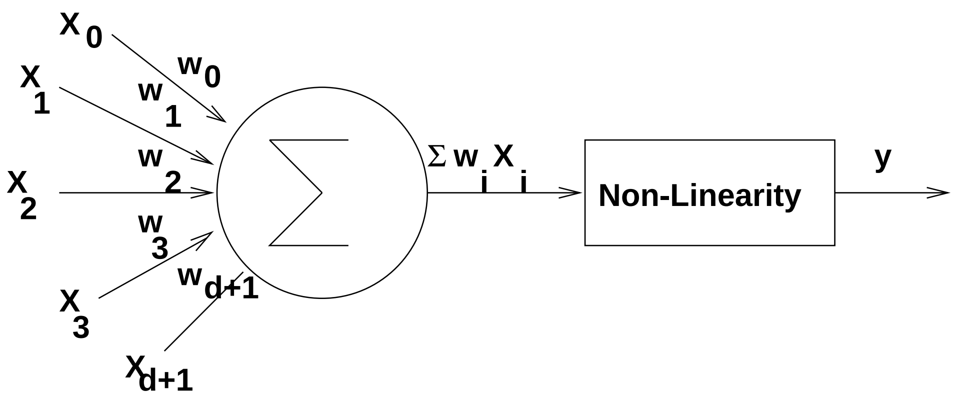


Figure 5.5: Schematic of a perceptron.

been built. A lookup table is used for the activation function. Bortos *et al.* [28] describe a smaller network implementation using less powerful Xilinx XC 3042 FPGAs. Their system supports 5 input nodes, 4 hidden nodes and 2 output nodes. A neuron is based on two XC 3042s and the nonlinearity is based on an 8K EPROM-based lookup table. Several other implementations have been surveyed in [161].

In implementing a neural network classifier on Splash 2, a perceptron implementation has been used as a building block. Hence, the design of a perceptron is described first. A perceptron consists of two stages namely, (i) an inner product computation, and (ii) a non-linear function applied to the output of the previous stage as shown in Figure 5.5. In our case, the perceptron is assumed to have 20 inputs which uses a non-linear function (typically a sigmoid function) to produce a real-valued output. We have used $tanh(\beta x)$ with $\beta = 0.25$ as the non-linearity in our implementation. For our mapping, two physical PEs serve as a neuron. The first PE handles the inner product phase and the second PE handles the non-linerity stage and writing result to the external memory operations. As the connection weights are fixed, an efficient way of handling the multiplication is to employ a lookup table. With a large external memory available at every PE, the lookup table can be stored. A pattern vector component is presented at every clock cycle. The PE looks up the multiplication table to obtain the weighted product and the sum is computed using an accumulator. Thus, after all the components of a pattern vector have been examined, we have computed the inner product. The non-linearity is again stored as a lookup table in the second PE. On receiving the inner product result from the first PE, the second PE uses the result as the address to the non-linearity lookup table and produces the output. Thus, the output of a neuron is obtained. The output is written back to the external memory of the second PE starting from a prespecified location. After sending all the pattern vectors, the host can read back the memory contents.

A layer in the neural net is nothing but a collection of neurons working synchronously on the input. On Splash 2, this can be easily achieved by broadcasting the input to as many physical PEs as desired. The output of the neuron is written into a specified segment of external memory and read back by the host at the end.

For every layer in the MLP, this exercise is repeated until the output layer is reached. Note that with change in the layers, the lookup table needs to be changed. Thus, we have been able to achieve a MLP on Splash 2 utilizing the hardware resources including the crossbar for broadcast purpose. A wavefront of computation proceeds one layer at a time. The schematic of the mapping for a single layer is shown in Figure 5.6.

The second design approach tries to exploit the onboard crossbar to achieve a single pass computation for both the layers (hidden and output). Thus it attempts to minimize the total number of clock cycles. In an ANN with n nodes in a layer,

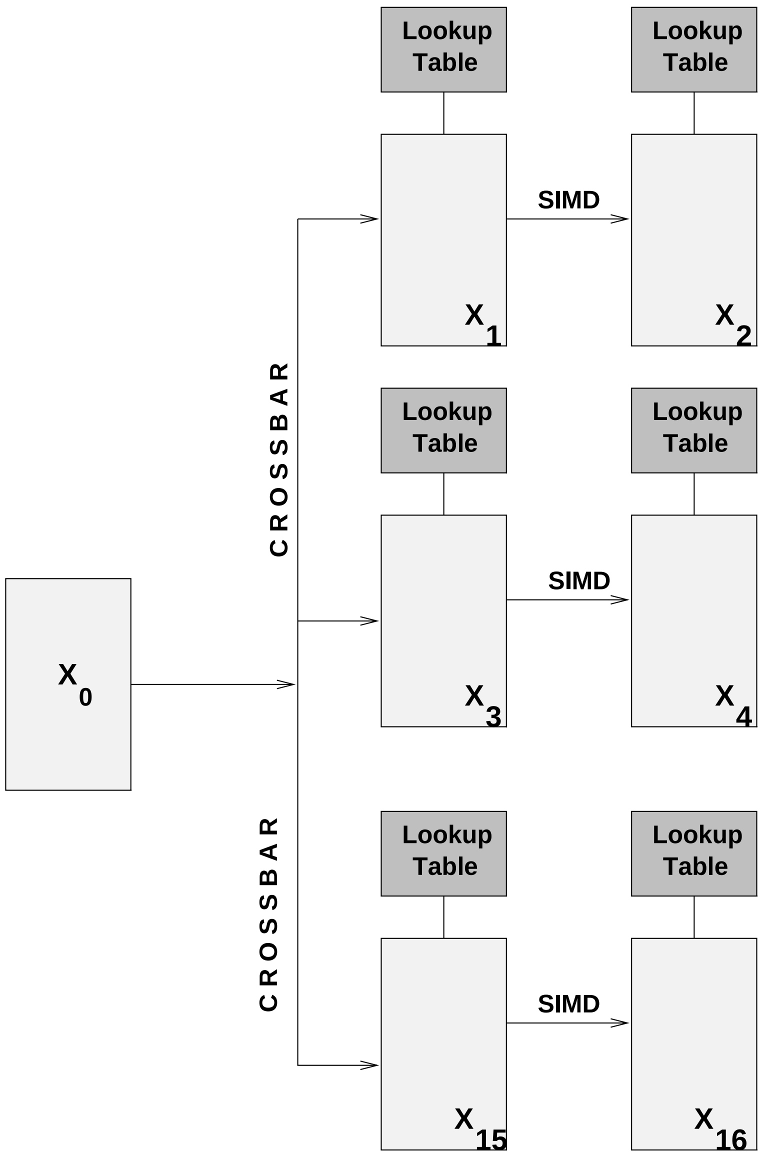


Figure 5.6: Mapping a single layer of a MLP onto Splash 2. The i^{th} PE computes $f(\sum w_{ij} F_i)$.

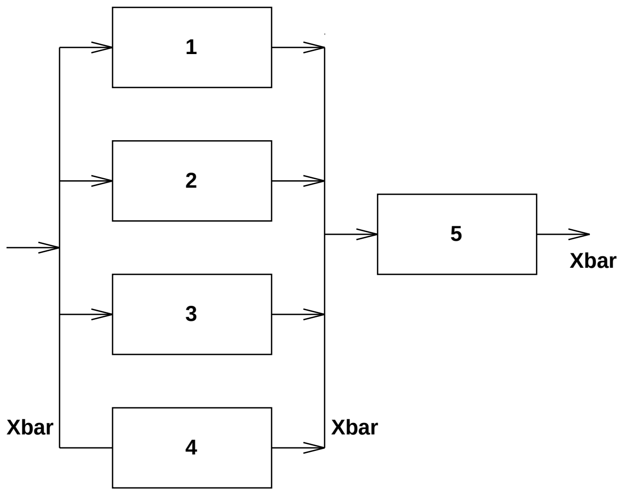


Figure 5.7: A modular building block for 3-layer MLP.

$O(n^2)$ data paths are required. A divide-and-conquer approach has been adopted in this algorithm to reduce the number of data paths. A modular 3-layer network is designed consisting of 4 hidden nodes and up to 3 output nodes as shown in Figure 5.7. The PEs 1 through 4 are the hidden layer neurons and the PE 5 serves as the physical PE for up to 3 virtual output PEs. For an n-dimensional input vector, an output is obtained every n cycles. PE 5 utilizes this to time multiplex the reading of result phase from the PEs 1 through 4. The rest of the cycles are used to perform several lookup operations (multiplications and non-linearity) for all the virtual PEs it handles. Thus, every time the PEs 1-4 are ready, PE 5 is ready with the result from the previous set. The result of several modules is cascaded through the use of a crossbar to obtain the final result. The PE mapping is shown in Figure 5.8.

The second model is not a general model but it is highly suitable when the dimension of the input pattern vectors is large and the number of outputs is small. It avoids reading and writing the intermediate results as in the earlier approach. Thus, it saves a large number of clock cycles as the number of input patterns is very large

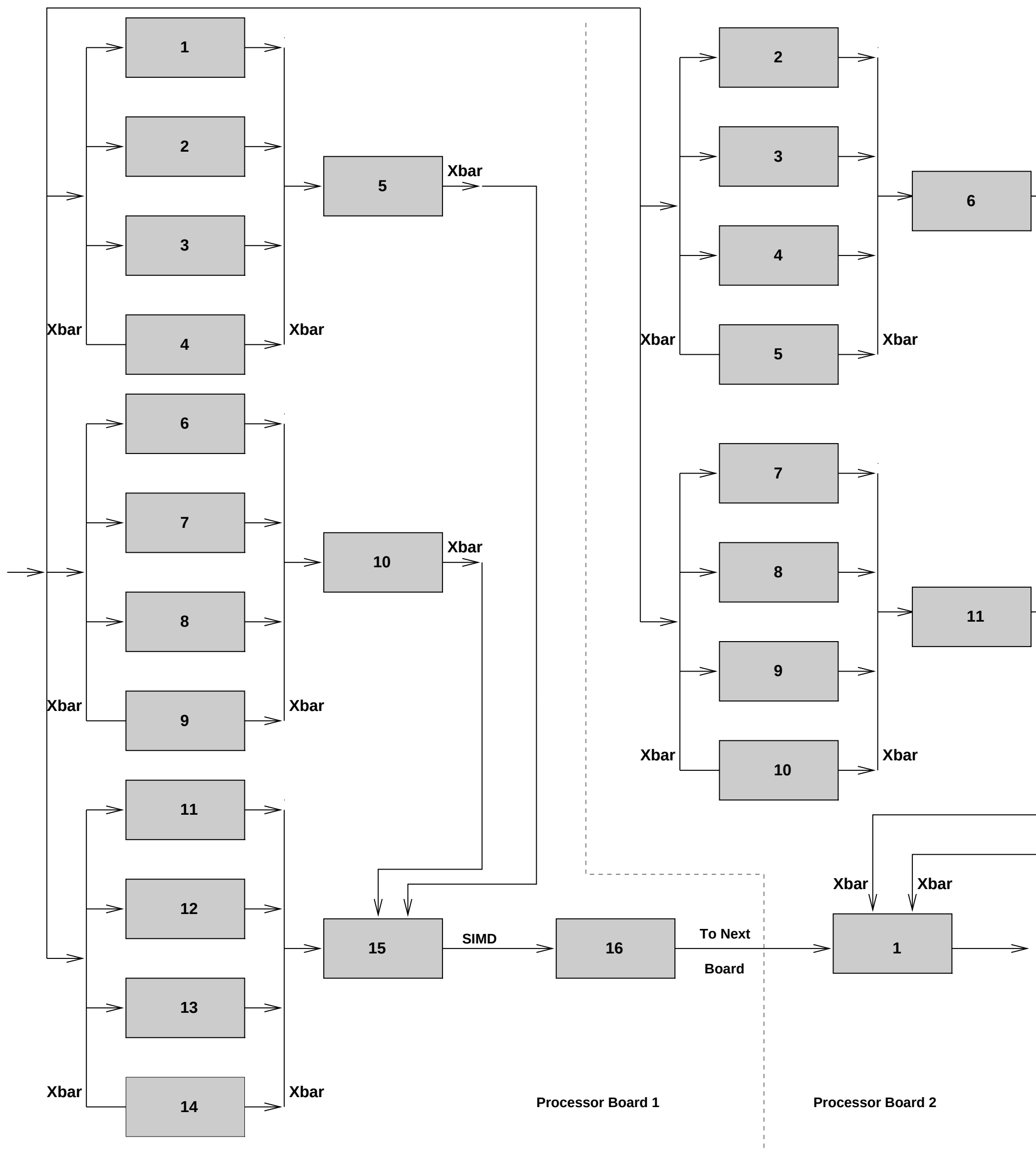


Figure 5.8: Overall PE mapping for implementing MLP.

in our case of page layout segmentation problem. The onboard crossbar is utilized to its maximum capability. The present crossbar on the processor board can store eight predefined configurations and switch every clock cycle if desired. The other feature of the crossbar is programmability at byte level. Both these features have been extensively used in the design. A typical configuration in a module with 4 neurons at the first level and 3 at the second level would use the following crossbar configuration.

```
--Crossbar configurations
-- Configuration 0
1 5 5 5 16 16 -- PE 1-4 receive 16-bits from PE 0
2 5 5 5 16 16
3 5 5 5 16 16
4 5 5 5 16 16
5 1 1 1 15 15 -- PE 5 receives input from PE 1 and outputs 16 bits to PE 15
-- Configuration 1
1 5 5 5 16 16 -- PE 1-4 receive 16-bits from PE 0
2 5 5 5 16 16
3 5 5 5 16 16
4 5 5 5 16 16
5 2 2 2 15 15 -- PE 5 receives input from PE 2  and outputs 16 bits to PE 15
-- Configuration 2
1 5 5 5 16 16 -- PE 1-4 receive 16-bits from PE 0
2 5 5 5 16 16
3 5 5 5 16 16
4 5 5 5 16 16
5 3 3 3 15 15 -- PE 5 receives input from PE 3 and outputs 16 bits to PE 15
-- Configuration 3
1 5 5 5 16 16 -- PE 1-4 receive 16-bits from PE 0
2 5 5 5 16 16
3 5 5 5 16 16
4 5 5 5 16 16
5 4 4 4 15 15 -- PE 5 receives input from PE 4 and outputs 16 bits to PE 15
-- Configuration 4
1 5 5 5 16 16 -- PE 1-4 receive 16-bits from PE 0
2 5 5 5 16 16
3 5 5 5 16 16
4 5 5 5 16 16
15 5 5 10 10 0 -- PE 15 receives input from PE 5 and PE 10
```

Due to present hardware design limitations, if PE 0 uses the crossbar then PE 16 cannot. PE 16 can communicate with the next processor board through the SIMD bus. Hence, PE 16 receives the input from the PE 15 using the SIMD bus and passes the partial results to PE 1 of the next board. This temporal parallelism of a crossbar is very helpful in realizing the large interconnection bandwidth requirement of neural nets.

5.2.4 Analysis of neural network implementation

For the first approach, let d denote the number of features (no. of input layer nodes), K be the number of patterns to be classified and l be the number of layers in the network. In our implementation, $d = 20$, $l = 2$ and K is the total number of pixels in the input image. The following analysis holds.

No. of clock cycles needed $= d * k * l$. For given values of d, K and l, the no. of clock cycles $= 20 * 2 * \; 10^6 = 40$ million.

With a clock rate of 22 MHz, time taken for 40 million clock ticks = 1.81 secs.

No. of PEs needed = No. of nodes in each layer.

For the case when the number of PEs required is larger than the available PEs, either more processor boards need to be added or the PEs need to be time shared. Note that the neuron outputs are produced independent of other neurons and the algorithm waits till the computations in the whole layer is completed.

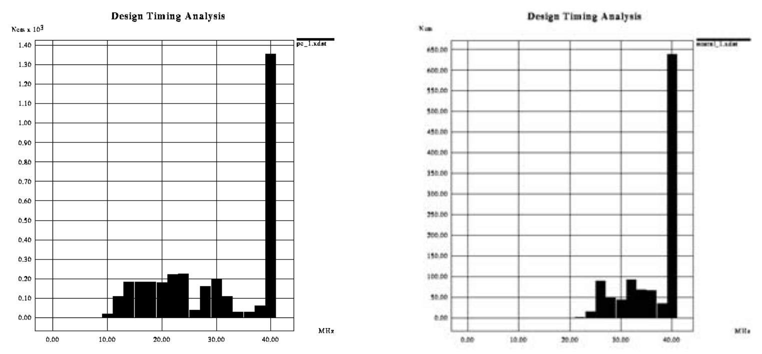
For the second approach, only a single pass through is needed. Hence, the number of clock cycles needed reduces linearly by the factor l (number of layers). Based on the above calculations, it will need only 0.9 sec to complete the task of classification of one million pixels.

5.2.5 Scalability

Both the implementations scale well with an increase in the number of input nodes. The second approach is limited by the path width on the crossbar. A MLP has a communication complexity of $O(n^2)$, where n is number of nodes. As n grows, it will be difficult to get good results from a single processor system. With a large number of processor boards, the single input data bus of 36-bits can cater to multiple input patterns. Note that in a multi-board system, all the boards receive the same input. This parallelism can give rise to more data streaming into the system, thus reducing the number of clock cycles by a linear factor. For a 12-bit input, the scale down factor is 3.

5.2.6 Speed evaluation for neural network implementation

For the present network with 20 input nodes, on a 2-board system, we achieve 176 million connections per second (MCPS) in a layer stage by running the clock at 22 MHz. In general, for a b-processor board system, a total speed of $176b$ MCPS is achievable. Thus, a 6-board system can deliver more than a billion connections per second.



(a) (b)

Figure 5.9: Synthesis speed of the two stages in segmentation algorithm. (a) Filtering; (b) Classification.

5.3 Analysis of the whole page layout algorithm

The page segmentation algorithm has been mapped onto a Splash 2 with 2 processor boards (i.e., at most 32 PEs are available for mapping). The functions of the PEs are modeled using VHDL. In our page segmentation algorithm, we have two main tasks: (i) filtering and (ii) classification using neural networks. Except for the host-interface development where C-language is used, the other two stages need VHDL-based designs. The simulation phase confirms the correctness of the algorithm. The results are verified using the timing diagrams obtained from the simulator. The synthesis speed for the filtering stage is 10.8 MHz as shown in Figure 5.9(a). Using *t*2, the synthesis results have been tested for correctness. We need 20 filters and we have 2 processor boards. Hence, we have to make a total of 10 passes over the input image. For this purpose, we use the reconfigurability of the FPGAs to change the instructions dynamically. Note that each filter has different mask values, so we need different sets of adders which change the instructions for the PEs. In terms of the number of operations per second, the clock speed reflects the rate at which the input pixels will be handled, i.e., 10.8 million pixels/second. Hence, a 1,024 $\times$ 1,024 image can be processed in approximately 0.1 seconds. Thus, 10 passes through an image would take approximately one second.

The synthesis speed for the neural network stage is projected at 22.0 MHz as shown in Figure 5.9(b). The neural network stage is expected to take 1 second using the second approach. Therefore, the total processing time on a 2-board system is 2.8 seconds using the first approach. However, the classification time can be reduced to 0.9 secs by using the second approach.

This computation time of two seconds for the segmentation algorithm can be compared with the computation time of 250 seconds needed on a SPARCstation 20 (33 MFlops). In other words, 20 billion operations are carried out in two seconds using the Splash 2 system.

5.4 Discussion

The mapping of this multi-stage algorithm brings out several advantages of the CCMs. The masks being large, the 40 MHz pipeline commercial convolvers can work only at 10 MHz. The Splash 2 synthesis speed of around 10 MHz did not put itself in any disadvantageous position. The twenty masks could be reprogrammed on single or multiple boards to provide a better throughput using the dynamic reconfigurability.

The interaction of the filtering and neural network-based classifier is also carried out using the same dynamic reconfigurability. Note that the whole processor board gets reconfigured for an application. This algorithm could potentially exploit many boards for the filtering stage. But, the neural network stage uses a maximum of two boards. Just like reprogramming each of the filters, the neural network stage is also considered as yet another stage for reconfigurable computing. As the control bit streams have been generated for each filter separately, any change in the filter mask values requires a resynthesis of that filter. Using the resources on the processor boards, this application demonstrates the MIMD mode of programming a Splash 2.

5.5 Summary

This chapter dealt with a mapping of specific chosen image segmentation algorithm onto Splash 2. The 2-D convolution operator defined in chapter 4 was used to achieve floating point image filtering. An multilayer feedforward neural network algorithm has been implemented. Two approaches for mapping a MLP on Splash 2 were presented and evaluated. An important attribute of the CCMs allows us to combine various sub-stages of an algorithm on the same system by just changing the control bit stream. This property is useful in designing real-time complex vision systems.

Chapter 6

Point Pattern Matching

Point pattern matching, i.e., finding the correspondence between two sets of points in an m-dimensional space, is a fundamental problem in many computer vision tasks. For example, feature-based rigid object recognition can be considered as an instance of point pattern matching. In motion and stereo analysis, point pattern matching is used to solve the correspondence problem. In remote sensing applications, point pattern matching is used for image registration.

For the general problem of point pattern matching, where no *a priori* knowledge about the two sets of points is available, a number of algorithms have been described in the literature [17, 222, 205, 9, 221, 217]. Baird's $O(n^2)$ algorithm, where n is the number of points in each of the two point sets, becomes more complex ($O(n^3)$) when the number of points in the two sets are not the same. Vinod et al. [222] propose a neural network for point pattern matching after formulating it as a 0-1 integer programming problem. A genetic algorithm has been suggested by Ansari et al. [9]. Most of these algorithms do not permit elastic distortion of the points, i.e., the points are assumed to have undergone a rigid body transformation.

The focus of this chapter is limited to the point pattern matching problem in the context of fingerprint matching. The two point sets can have different numbers of points. We do not currently handle scaling and rotation in the point sets, but allow elastic distortion. In fingerprint matching, we are interested in the set of "paired features" between the query fingerprint and database (reference) fingerprints. This process is repeated over all the records in the fingerprint database. Because of the large size of the fingerprint database, special hardware accelerators are needed for matching. Due to the elasticity of the skin and non-ideal nature of the imaging process in capturing the fingerprint impressions, distortions of the feature vectors are inevitable.

6.1 Fingerprint matching

Fingerprint-based personal identification is the most popular biometric technique used in automatic personal identification [156]. Law enforcement agencies use it routinely for criminal identification. Now, it is also being used in several other applications such as access control for high security installations, credit card usage verification, and employee identification [156]. The main reason for the popularity of fingerprints as a form of identification is that the fingerprint of a person is unique and the features used for matching remain invariant through age.

A fingerprint is characterized by ridges and valleys. The ridges and valleys alternate, flowing locally in a constant direction (see Figure 6.1). A closer analysis of the

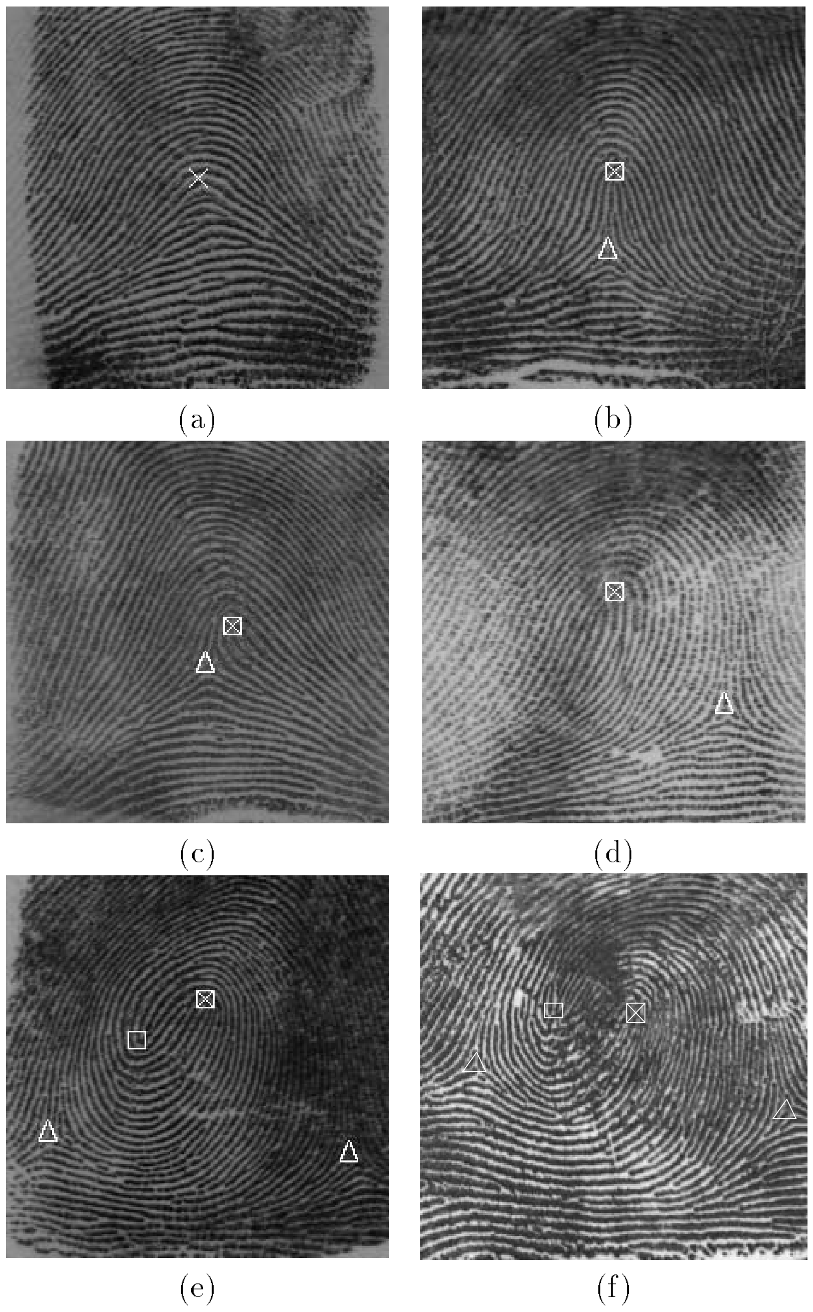


Figure 6.1: Gray level fingerprint images of different types of patterns with core (□) and delta (△) points: (a) arch; (b) tented arch; (c) right loop; (d) left loop; (e) whorl; (f) twin loop.

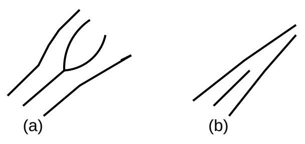


Figure 6.2: Two commonly used fingerprint features: (a) Ridge bifurcation; (b) Ridge ending.

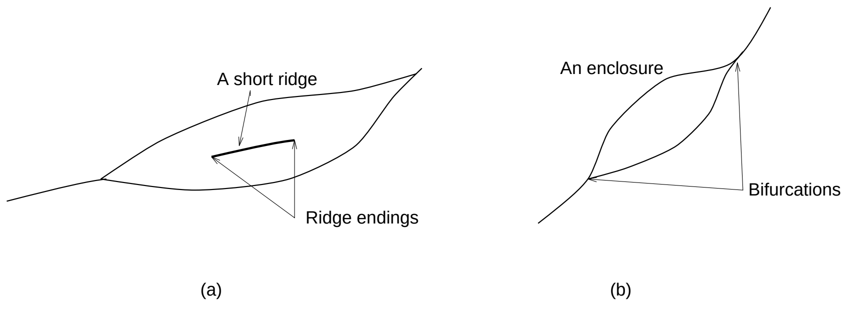


Figure 6.3: Complex features as a combination of simple features: (a) Short ridge; (b) Enclosure.

fingerprint reveals that the ridges (or the valleys) exhibit anomalies of various types, such as ridge bifurcations, ridge endings, short ridges, and ridge crossovers. Eighteen different types of fingerprint features have been enumerated in [65]. Collectively, these features are called *minutiae*. For automatic feature extraction and matching, only two types of minutiae are considered: ridge endings and ridge bifurcations.

Ridge endings and bifurcations are shown in Figures 6.2(a) and 6.2(b). No distinction between these two feature types is made during matching since data acquisition conditions such as inking, finger pressure, and lighting can easily change one type of feature into another. More complex fingerprint features can be expressed as a combination of these two basic features. For example, a short ridge (see Figure 6.3(a)) can be considered as a collection of a pair of ridge endings, and an enclosure (see Figure 6.3(b)) can be considered as a collection of two bifurcations.

A survey of commercially available automatic fingerprint identification systems

(AFIS) is available in [132]. Well-known manufacturers of automatic fingerprint identification systems include NEC Information Systems, De La Rue Printrak, North American Morpho, and Logica. In order to provide a reasonable response time for each query, commercial systems use dedicated hardware accelerators or application-specific integrated circuits (ASICs).

An automatic fingerprint identification system (AFIS) consists of various processing stages as shown in Figure 6.4. For the purpose of automation, a suitable representation (feature extraction) of fingerprints is essential. This representation should have the following desirable properties:

1. Retain the discriminating power (uniqueness) of each fingerprint at several levels of resolution (detail).
2. Easily computable.
3. Amenable to automated matching algorithms.
4. Stable and invariant to noise and distortions.
5. Efficient and compact representation.

The compactness property of representation often constrains its discriminating power. Clearly, the input digital image of a fingerprint itself does not meet these representational requirements. Hence, high-level structural features are extracted from the image for the purpose of representation and matching.

The commercially available fingerprint systems typically use ridge bifurcations and ridge endings as features (see Figure 6.2). Because of the large size of the fingerprint

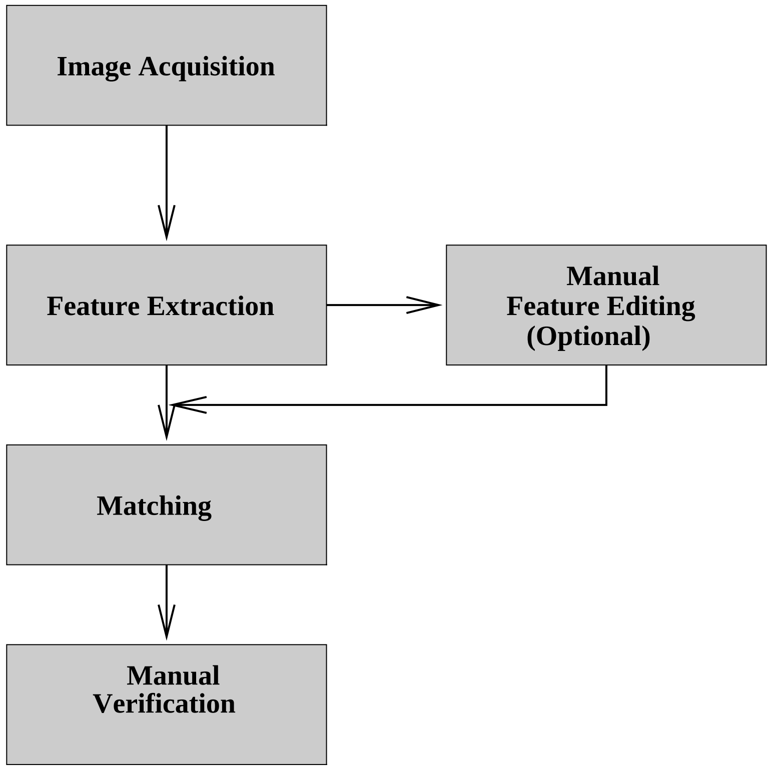


Figure 6.4: Stages in an AFIS

database and the noisy fingerprints encountered in practice, it is very difficult to achieve a reliable one-to-one matching in all test cases. Therefore, AFIS provides a ranked list of possible matches (usually the top ten matches) which are then verified by a human expert. The matching stage uses the position and orientation of the ridge at the minutiae point. Therefore, reliable and robust extraction of minutiae points can simplify the matching algorithm and obviate the manual verification stage.

One of the main problems in extracting fingerprint features is the presence of noise in the fingerprint image. Commonly used methods for taking fingerprint impressions involve applying a uniform layer of ink on the finger and rolling the finger on paper. This leads to the following problems. Smudgy areas in the image are created by over-inked areas of the finger, while breaks in ridges are created by under-inked areas.

Additionally, the elastic nature of the skin can change the positional characteristics of the minutiae points depending on the pressure applied on the fingers. Although inkless methods for taking fingerprint impressions are now available, these methods still suffer from the positional shifting caused by the skin elasticity. The AFIS used for criminal identification poses yet another problem. Non-cooperative attitude of suspects or criminals in providing the impressions leads to a smearing of parts of the fingerprint impression. Thus, noisy features are inevitable in real fingerprint images. The matching module must be robust to overcome the noisy features generated by the feature extraction module.

The feature extraction process takes the input fingerprint gray-level image and extracts the minutiae features described earlier, making no efforts to distinguish between the two categories (ridge endings and ridge bifurcations). In this section, an algorithm for matching rolled fingerprints against a database of rolled fingerprints is presented. A query fingerprint is matched with every fingerprint in the database, discarding candidates whose matching scores are below a user-specified threshold. Rolled fingerprints usually contain a large number of minutiae (between 50 and 100). Since the main focus of this section is on the matching algorithm, we assume that the features (minutiae points) have already been extracted from the fingerprint images. In particular, we assume that the core point of the fingerprint is known and that the fingerprints are oriented properly. This implies that the fingerprints have been approximately registered.

Matching a query and a database fingerprint is equivalent to matching their minutiae sets. Each query fingerprint minutia is examined to determine whether there is

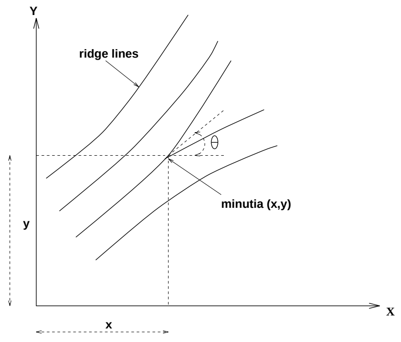


Figure 6.5: Components of a minutia feature.

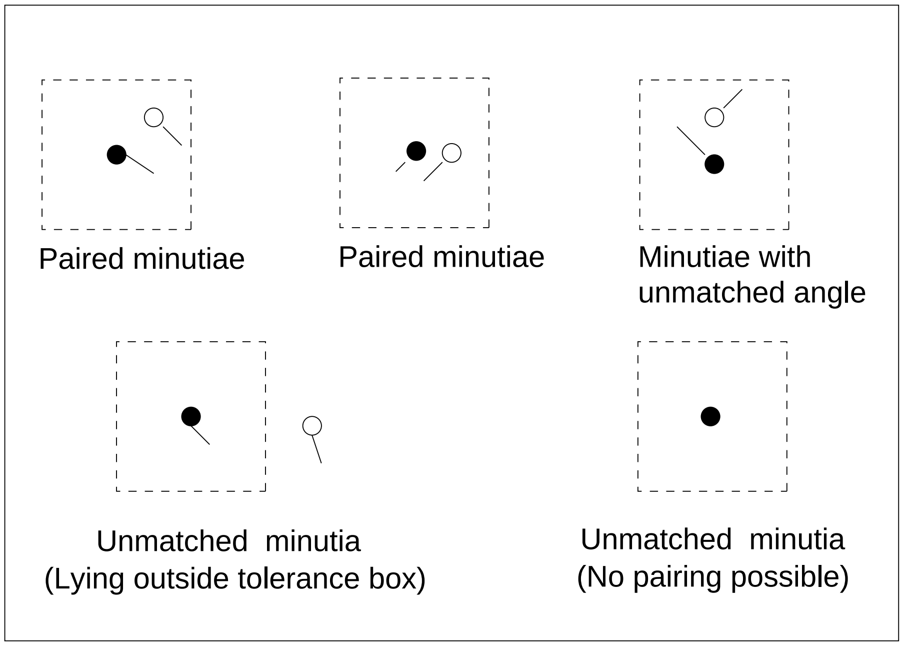
a corresponding database fingerprint minutia. A feature vector is characterized by its three components (x, y, θ) as shown in figure 6.5. Two minutiae are said to be *paired or matched* if their components (x, y, θ) are in "close" proximity to each other. Following three situations arise as shown in Figure 6.6.

1. A database fingerprint minutia matches the query fingerprint minutia in all the components (paired minutiae);

2. A database fingerprint minutia matches the query fingerprint minutia in the x and y coordinates, but does not match in the direction component (minutiae with unmatched angle);

3. No database fingerprint minutia matches the query fingerprint minutia (unmatched minutia).

Of the three cases described above, the minutiae are said to be paired only in the first case.


Paired minutiae
Paired minutiae
Minutiae with unmatched angle
Unmatched minutia (Lying outside tolerance box)
Unmatched minutia (No pairing possible)

⬚	- Tolerance box
●	- Query fingeprint minutia
○	- Database fingerprint minutia

Figure 6.6: Possible outcomes in minutia matching.

6.2 Matching algorithm

The following notation is used in the sequential and parallel matching algorithms described below. Let the query fingerprint be represented as an n-dimensional feature vector $\mathbf{f^q} = (\mathbf{f^q_1},\mathbf{f^q_2},......,\mathbf{f^q_n})$. Note that each of the n elements is a feature vector corresponding to one minutia, and the i^{th} feature vector $\mathbf{f_i}$ contains three components, $\mathbf{f_i} = (f_i(x), f_i(y), f_i(\theta))$.

The components of a feature vector are shown geometrically in Figure 6.5. The query fingerprint core point is located at (C^q_x, C^q_y). Similarly, let the r^{th} reference (database) fingerprint be represented as an m_r-dimensional feature vector

$\mathbf{f^r} = (\mathbf{f^r_1},\mathbf{f^r_2},.....,\mathbf{f^r_{m_r}})$, and the reference fingerprint core point is located at (C^r_x, C^r_y).

Let (x^t_q, y^t_q) and (x^b_q, y^b_q) define the bounding box for the query fingerprint, where x^t_q

is the x-coordinate of the upper left corner of the box and x_q^b is the x-coordinate of the lower right corner of the box. Quantities y_q^t and y_q^b are defined similarly. A bounding box is the smallest rectangle that encloses all the feature points. Note that the query fingerprint $\mathbf{f^q}$ may or may not belong to the fingerprint database $\mathbf{f^D}$. The fingerprints are assumed to be registered with a known orientation. Hence, there is no need of normalization for rotation. The matching algorithm is based on finding the number of paired minutiae between each database fingerprint and the query fingerprint. It uses the concept of minutiae matching described earlier. In order to reduce the amount of computation, the matching algorithm takes into account only those minutiae that fall within a common bounding box. The common bounding box is the intersection of the bounding box for the query and reference (database) fingerprints. Once a count of the matching minutiae is obtained, a matching score is computed. The matching score is used for deciding the degree of match. Finally, a set of top scoring reference fingerprints is obtained as a result of matching.

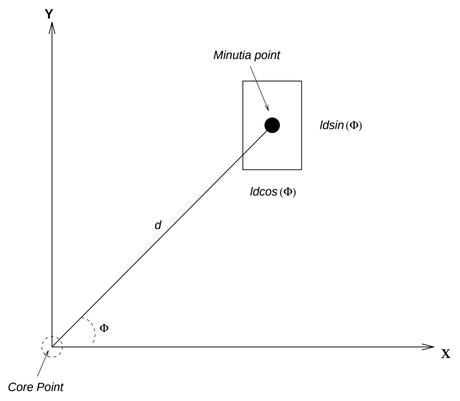


Figure 6.7: Tolerance box for X- and Y-components.

The sequential matching algorithm is described in Figure 6.8. In the sequential algorithm, the tolerance box (shown in Figure 6.7 with respect to a query fingerprint

```
Input: Query feature vector $\mathbf{f^q}$ and the rolled fingerprint database $\mathbf{f^D}=\{\mathbf{f^r}\}_{r=1}^{N}$.
       The $r^{th}$ database fingerprint is represented as an $m_r$-dimensional feature vector
       and the query feature vector is n-dimensional.
Output: A list of top ten records from the database with matching scores $> T$.
Begin
    For r=1 to N do
        1. Register the database fingerprint with respect to the core point $(C_x^q, C_y^q)$
        of the query fingerprint:
            For i=1 to $m_r$ do
                $f_i^r(x) = f_i^r(x) - C_x^q$
                $f_i^r(y) = f_i^r(y) - C_y^q$
        2. Compute the common bounding box for the query and reference fingerprints:
            Let $(x_q^t, y_q^t)$ and $(x_q^b, y_q^b)$ define the bounding box for the query fingerprint.
            Let $(x_r^t, y_r^t)$ and $(x_r^b, y_r^b)$ define the bounding box for the $r^{th}$
            reference fingerprint. The intersection of these two boxes is the
            common bounding box. Let the query print have $M_e^q$ and
            reference print have $N_e^r$ minutiae in this box.
        3. Compute the tolerance vector for $i^{th}$ feature vector $f_i^r$:
            If the distance from the reference core point to the current reference feature
            is less than K then
                $t_i^r(x) = ldcos(\phi)$,
                $t_i^r(y) = ldsin(\phi)$, and
                $t_i^r(\theta) = k_3$,
            else
                $t_i^r(x) = k_1$,
                $t_i^r(y) = k_2$, and
                $t_i^r(\theta) = k_3$,
                where $l$, $k_1$, $k_2$ and $k_3$ are prespecified constants determined
                empirically based on the average ridge width,
                $\phi$ is the angle of the line joining the core point
                and the $i^{th}$ feature with the x-axis,
                and $d$ is the distance of the feature from the core point.
                Tolerance box is shown geometrically in Figure 6.7.
        4. Match minutiae:
                Two minutiae $\mathbf{f_i^r}$ and $\mathbf{f_j^q}$ are said to match if
                the following conditions are satisfied:
                $f_j^q(x) - t_i^r(x) \leq f_i^r(x) \leq f_j^q(x) + t_i^r(x)$,
                $f_j^q(y) - t_i^r(y) \leq f_i^r(y) \leq f_j^q(y) + t_i^r(y)$, and
                $f_j^q - t_i^r(\theta) \leq f_i^r(\theta) \leq f_j^q(\theta) + t_i^r(\theta)$,
                where $t_i^r = (t_i^r(x), t_i^r(y), t_i^r(\theta))$ is the tolerance vector.
            Set the number of paired features, $m_p^r = 0$;
            For all query features $\mathbf{f_j^q}$, j=1,2, ... $M_e^q$, do
                If $\mathbf{f_j^q}$ matches with any feature in $\mathbf{f_i^r}$, i=1,2, ..., $N_e^r$,
                then increment $m_p^r$. Mark the corresponding feature in $\mathbf{f^r}$ as paired.
        5. Compute the matching score (MS (q,r)):
                $MS(q,r) = \frac{m_p^r * m_p^r}{(M_e^q * N_e^r)}$.
    Sort the database fingerprints and obtain top 10 scoring database fingerprints.
End
```

Figure 6.8: Sequential fingerprint matching algorithm.

minutia) is calculated for the reference (database) fingerprint minutia. In the parallel algorithm described in the next chapter, the tolerance box is calculated for the query fingerprint. A similar sequential matching algorithm is described in [227]. Depending on the desired accuracy, more than one finger could be used in matching. In that case, a composite score is computed for each set.

6.3 Mapping point pattern matching onto Splash 2

We parallelize the matching algorithm so that it utilizes the specific characteristics of the Splash 2 architecture. While performing this mapping, we need to take into account the limitations of the available FPGA technology. Any preprocessing needed on the query minutiae set is an one-time operation, whereas reference fingerprint minutiae matching is a repetitive operation. Computing the matching score involves a floating point division. The floating point operations and one-time operations are performed in software on the host whereas the repetitive operations are delegated to the Splash 2 PEs . The parallel version of the algorithm involves operations on the host, on X_0, and on each PE.

One of the main constructs in the parallel point matching algorithm is a lookup table which consists of all possible points within the tolerance box around a feature vector. The Splash 2 data paths for the parallel algorithm are shown in Figure 6.9. The host processes the query and database fingerprints as follows. The query finger-

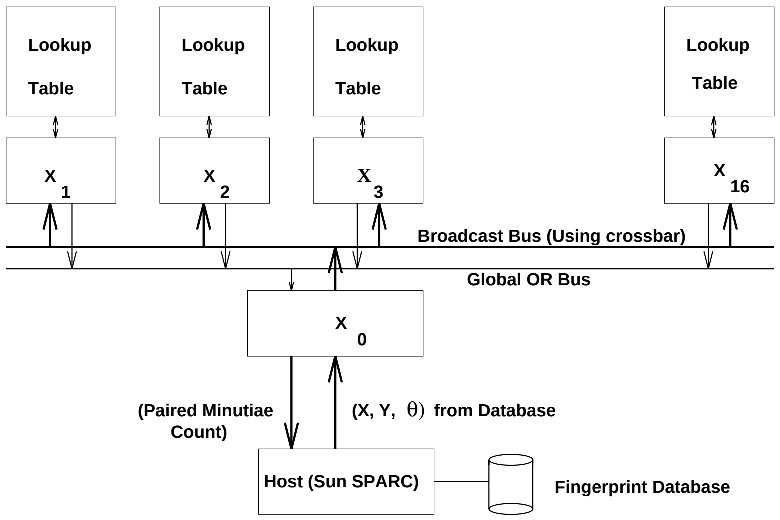


Figure 6.9: Data flow in parallel point matching algorithm.

print is read first and the following preprocessing is done:

1. The core point is assumed to be available.

   For the given query feature $\mathbf{f^q}$, generate a tolerance box. Enumerate a total of $(t_x \times t_y \times t_\theta)$ grid points in this box, where t_x is the tolerance in x, t_y is the tolerance in y and t_θ is tolerance in θ.

2. Allocate each feature to one PE in Splash 2. Repeat this cyclically, i.e., features 1-16 are allocated to PEs X_1 to X_{16}, features 17-32 are allocated to PEs X_1 to X_{16}, and so on.

3. Initialize the lookup tables by loading the grid points within each tolerance box in step (1) into the memory.

In this algorithm, the tolerance box is computed with respect to the query fingerprint features. The host then reads the database of fingerprints and sends their feature vectors for matching to the Splash 2 board.

For each database fingerprint, the host performs the following operations:

1. Reads the feature vectors.
2. Registers the features as described in step (1) of the sequential algorithm in Figure 6.8.
3. Sends each of the feature vectors over the Broadcast Bus to all the PEs if it is within the bounding box of the query fingerprint.

For each database fingerprint, the host then reads the number of paired features m_p^r that was computed by the Splash 2 system, $r = 1, \ldots N$. Finally, the matching score is computed as in the sequential method.

6.3.1 Computations on Splash 2

The computations carried out on each PE of Splash 2 are described below. As mentioned earlier, X_0 plays a special role in controlling the crossbar in Splash 2.

1. Operations on X_0:

   Each database feature vector received from the host is broadcast to all the PEs. If it is matched with a feature in a lookup table, then the PE drives the Global OR Bus high. When the OR Bus is high, X_0 increments a counter. The host reads this counter value (m_p^r) after all the feature vectors for the current database fingerprint have been processed. Operations on X_0 are highlighted in Figure 6.10.

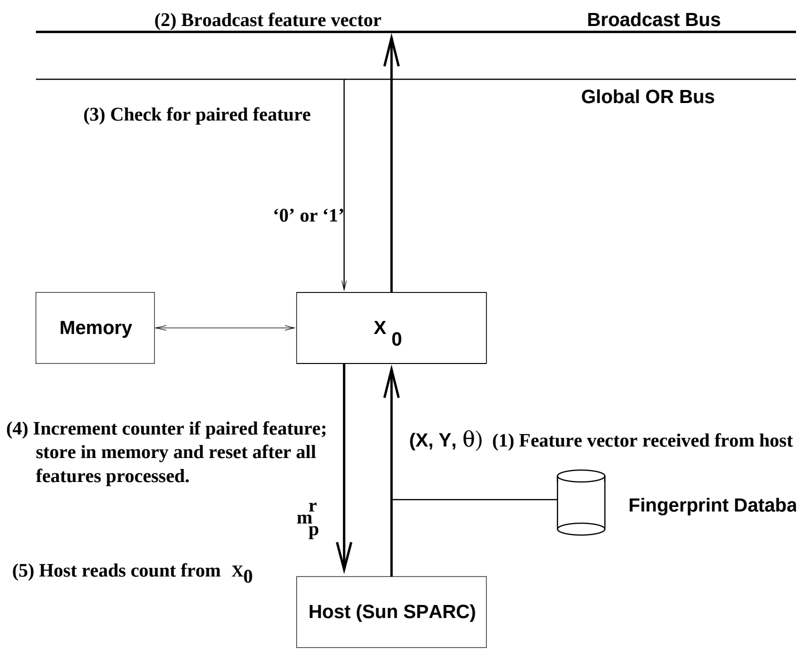

(2) Broadcast feature vector
Broadcast Bus
Global OR Bus
(3) Check for paired feature
'0' or '1'
Memory
X 0
(4) Increment counter if paired feature; store in memory and reset after all features processed.
(X, Y, θ) (1) Feature vector received from host
Fingerprint Databa
m r p
(5) Host reads count from X0
Host (Sun SPARC)

Figure 6.10: Data flow in X_0.

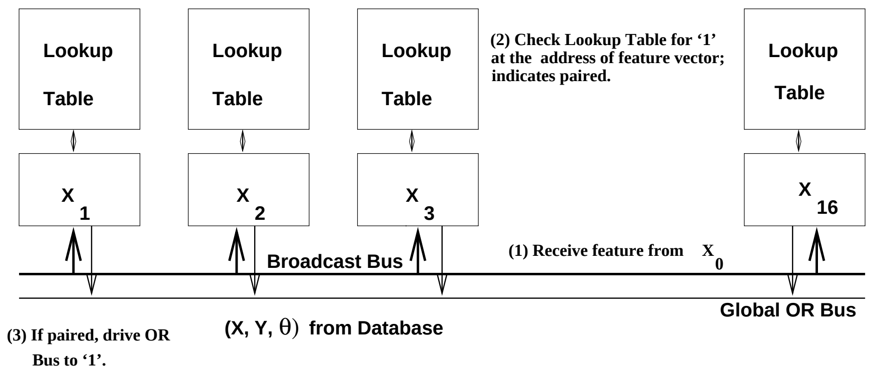

Lookup Table
Lookup Table
Lookup Table
(2) Check Lookup Table for '1' at the address of feature vector; indicates paired.
Lookup Table
X 1
X 2
X 3
X 16
Broadcast Bus
(1) Receive feature from X 0
Global OR Bus
(3) If paired, drive OR Bus to '1'.
(X, Y, θ) from Database

Figure 6.11: Data flow in a PE.

2. Operations on each PE:

   On receiving the broadcasted feature, a PE computes its address in the lookup table through a hashing function. If the data at the computed address is a '1', then the feature is paired, and the PE drives the Global OR Bus high. Operations on a PE are highlighted in Figure 6.11.

6.4 Analysis of point pattern matching algorithm on Splash 2

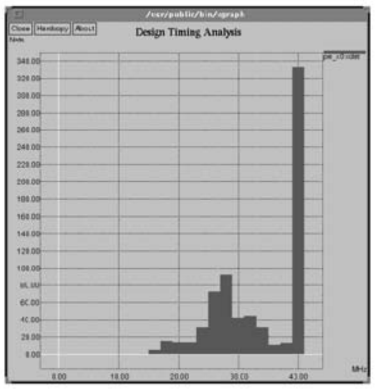
The analysis of the parallel implementation is carried out in two respects: (i) simulation and synthesis results, and (ii) speed.

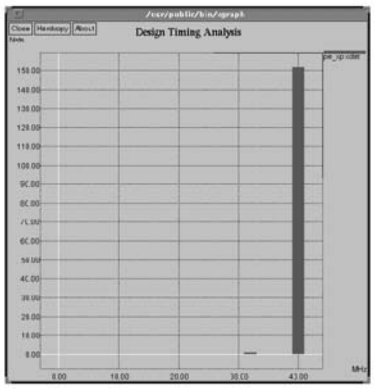
6.4.1 Simulation and synthesis results

The VHDL behavioral modeling code for PEs X_0 – X_{16} has been tested using the Splash simulation environment. The simulation environment loads the lookup tables and crossbar configuration file into the simulator. Note that the Splash simulator runs independent of the Splash 2 hardware and runs on the host. The input data are read from a specified file, and the data on each of the signals declared in the VHDL code can be traced as a function of time.

The synthesis process starts by translating the VHDL code to a Xilinx netlist format (XNF). The vendor-specific 'ppr' utility generates placement, partitioning, and routing information from the XNF netlist. The final bit stream file is generated using the utility 'xnf2bit'. The 'timing' utility produces a graphical histogram of the speed at which the logic can be executed. The logic synthesized for X_0 can run at a clock rate of 17.1 MHz, and the logic for the PEs X_1 to X_{16} can run at 33.8 MHz. Observe that these clock rates correspond to the longest delay (critical) paths, even though most of the logic could be driven at higher rates. Increased processing speed may be possible by optimizing the critical path.

The bit stream files for Splash 2 are generated from the VHDL code. Using the





(a) (b)

Figure 6.12: Speed projections. (a) X_0; (b) other X_is

C interface for Splash 2, a host version of the fingerprint matching application is generated. The host version reads the fingerprint database from the disk and obtains the final list of candidates after matching.

6.4.2 Performance analysis

The sequential algorithm, described in Section 6.2, executed on a Sun SPARCstation 20 performs at the rate of 100 matches per second on database and query fingerprints that have approximately 65 features. A match is the process of determining the matching score between a query and a reference fingerprint. The Splash 2 implementation should perform matching at the rate of 2.6×10^5 matches per second. This matching speed is obtained from the 'timing' utility. The host interface part can run at 17.1 MHz and each PE can run at 33.8 MHz. The speed graphs obtained from the 'timing' utility are shown in Figure 6.12. Hence, the entire fingerprint matching will run at the slower of the two speeds, i.e., 17.1 MHz. Assuming a total of 65 minutiae, on an average, in a database fingerprint, the matching speed is estimated at 2.6×10^5 matches per second. We evaluated the matching speed using a database of 10,000 fingerprints created from 100 real fingerprints by randomly deleting, adding and perturbing minutiae. The measured speed on a Splash 2 system running at 1 MHz is of the order of 6,300 matches per second. The prototype Splash 2 system which is available to us has been run at 1 MHz clock rates involving data transfer from host through the SIMD bus. Assuming a linear scaling of performance with an increase in clock rate, we would achieve approximately 110,000 matches per second. We feel that the disparity in the projected and achieved speeds (2.6×10^5 versus 1.1×10^5) is due to different tasks being timed. The time to load the data buffers onto Splash 2 has not been taken into account in the projected speed, whereas this is included in the time measured by the host in an actual run.

The Splash 2 implementation is more than 1,100 times faster than a sequential implementation on a SPARCstation 20. Another advantage of the parallel implementation on Splash 2 is that the matching speed is independent of the number of minutiae in the query fingerprint. The number of minutiae affects only the lookup table initialization, which is done during preprocessing by the host, and this time is amortized over a large number of database records.

The matching algorithm will scale well as the number of Splash 2 boards on the system is increased. Multiple query fingerprints can be loaded on different Splash 2 boards, each matching against the database records as they are transferred from the host. This would result in a higher throughput from the system.

The processing speed can be further improved by replacing some of the soft macros on the host interface part (X_0) by hard macros. To sustain the matching rate, the data throughput should be at a rate of over 250,000 fingerprint records per second (with an average of 65 minutiae per record). This may be a bottleneck for the I/O subsystem.

6.5 Discussion

The mapping of the elastic point pattern matching algorithm on Splash 2 brings out the versatility of CCMs in getting reconfigured for any level of parallelism. This mapping also highlights benefits of translating computations to hashing-based lookups. In fact, the gain in the speed of the matching algorithm can be attributed to performing arithmetic computations using lookup tables. The mapping could potentially use many processor boards to improve the throughput of the whole system in addressing multiple queries. The point matching algorithm used the processor boards in a SIMD fashion for all the PEs except X_0. Technically, this algorithm mapping uses a mix of MIMD and SIMD processing modes.

6.6 Summary

In this chapter a high-level vision algorithm of point pattern matching was introduced. Its mapping on Splash 2 was described and the performance analyzed. When applied to fingerprint feature vector matching, a significant speedup has been observed. The main idea behind the mapping is to utilize lookup tables. In contrast to other mapping algorithms, this algorithm seeks a significant help from the host. Though a formal hardware-software codesign technique has not been applied, the principles of the task partitioning between Splash 2 (hardware) and software (host) have been similar to those used in codesign techniques.

Chapter 7

Building a Taxonomy of Computer Architectures

Custom computing machines (CCMs) differ from general-purpose processors and application specific integrated circuits (ASICs) in several ways. By constructing a taxonomy of the available architectures to build an embedded system, one can better understand their similarities and dissimilarities. This chapter aims at building a taxonomy of processors/co-processors based on many commonly observed features and performance measures for several CCMs and other well known compute engines. A multi-dimensional data analysis technique of hierarchical clustering is applied to construct a taxonomy of several platforms. Both the single-link and complete-link cluster-based taxonomies provide an appropriate way to classify the platforms.

Organizing objects into a taxonomy is an important step in the development of science and technology, especially as an aid to paradigmatic clarity and the development of prescriptive terminology [231]. Consider the range of systems available for information processing applications. A general-purpose uniprocessor such as the PowerPC, P6 or Alpha 21164 has a pre-designed instruction set which is used to write programs for given applications. A new application can be written by rearranging the sequence of instructions. However, the performance of these applications on the general-purpose processors is always limited in several ways, e.g., every instruction needs to be fetched and decoded before execution. In contrast, an application-specific integrated circuit (ASIC) provides an 'optimal' performance for the problem for which it has been designed, but lacks the flexibility of being used for any other application. This trade off between generality vs. performance has been observed to be the main driving force in the development of field programmable gate array (FPGA)-based custom computing machines (FCCMs)[32]. Several FCCMs have been built [32] and many applications have been designed using them with performance that is often comparable to supercomputers. A designer has this new option of using a FCCM in his embedded system designs. Often, the information about what is an FCCM and how does it differ from a general-purpose processor, an ASIC or a supercomputer is not available to the designer. In the spectrum of processor/co-processors, FCCMs have not yet been placed at their appropriate location. In this chapter, an effort to appropriately place FCCMs in the domain of processors is made by building a taxonomy.

Several taxonomies of computer architecture and processors exist in the literature. For example, based on the instruction set, a uniprocessor can be classified as a RISC or a CISC processor. Based on Flynn's taxonomy [96], processors have been classified on the basis of their instruction and data streams into four classes, namely, SISD, SIMD, MISD and MIMD. For MIMD parallel processors, Bell's taxonomy [96] is based on message passing or shared memory. Parallel processors can also be classified as fine-grained or coarse-grained depending on the type of processing elements and their interconnection capabilities.

A classification scheme can use benchmark results to characterize the hardware systems. Several benchmarks (e.g., Whetstone, Dhrystone) have been used to characterize complex computing systems. The problem with using benchmarks is that they are restricted to a class of machines which are "similar" to each other. For example, MIPS rating does not reflect the floating point capabilities of a processor. Many other indices such as sizeup [207], redundancy, utilization and quality of parallelism have been defined in [96]. For a VLSI designer, the performance criteria are quite different. The factors for comparing different VLSI systems can include (i) the technology used (e.g., nMOS, CMOS, ECL), (ii) silicon area, and (iii) speed. Many other criteria such as packing density or number of I/O pins are often employed. Like uniprocessors, FPGAs can not be characterized by a single index. Number of equivalent gates, number of I/O pins, and number of CLBs have been used in the past. Recently, many vendors have decided to accept PREP benchmark results as a performance evaluation criterion. The PREP benchmark suite consists of designing and implementing several standard digital circuits and measuring the performance in terms of capacity and speed. More details about the PREP benchmark tests are available in [110].

Most of the available taxonomies are typically based on fewer than four factors or features of the systems. A complex system is based on a large number of subsystems, hence a small number of features do not always reflect the overall characteristics of the system. Borrowing ideas from exploratory analysis of multi-dimensional data, we have built a complete taxonomy involving CCMs. A wide range of hardware platforms have been chosen and several factors characterizing these systems have been identified. Using a hierarchical clustering technique, a taxonomy has been developed. The technique used has the capability to handle a very large number of features.

CCM is a new concept and it needs to be compared and contrasted with existing concepts like parallel processors, uniprocessors, and special-purpose processors. Conceptual clustering arranges objects into classes representing certain descriptive concepts using symbolic and numerical attributes in contrast to numerical clustering where the features take only numerical values and a numerical distance measure is computed between a pair of patterns. Michalski and Stepp [154] proposed the use of conceptual clustering to build automated classification trees. They classified several computer systems based on the attributes used for describing them. Levine [139] describes a method of classifying several sports based on their attributes. Although Michalski and Stepp [154] argue superiority of conceptual clustering, Srivastava and Murthy [206] have shown the equivalence of conceptual clustering with conventional numerical clustering. Hence, we use numerical clustering techniques in our experiments.

7.1 Proposed method

In order to carry out a comparative analysis across a wide range of platforms based on different computing paradigms and architectures, the following set of twelve features are defined. These features describe the characteristics of the systems. The feature values have been chosen on a scale of 1 to 10 for relative comparison. This 12-dimensional feature space will be used to represent the performance of various architectures for our analysis. The features and their range of values is explained below. As some of these features and assigned values (the author assigned these values) are subjective, an interactive tool can be developed to experiment with different feature sets and values.

1. Programmability: Poor instruction set (e.g., a single instruction 'go') – 1; fixed instruction set – 5; user programmable instruction – 10;

2. Ease of use: Difficult to use – 1; Medium difficulty of use – 5; Easy to use – 10.

3. Speed of computation: General purpose uniprocessors – 3; General-purpose parallel processors–5; Vector processors – 5; Special-purpose parallel processors – 6; FPGAs – 7; ASICs – 10.

4. Modes of parallelism: Pipelined – 2; Vector Processing support – 4; SIMD – 5; MIMD – 7; Flexible – 10;

5. Scalability: ASICs – Not Applicable (1); Limited (Uniprocessors, Fixed-system boards) – 2; Special purpose parallel processor – 6; Reconfigurable logic arrays – 8; General Purpose Parallel Processors – 10.

6. Design Cycle time: Long: 1; Medium – 5; Short – 8; Fast – 10.

7. Inter-processor Communication: Uniprocessors – Not applicable (1); Crossbar+Systolic+limited off-board communication – 6; Multi-stage Interconnection Network – 8; Hierarchical – 8; Crossbar – 10.

8. Hardware support for floating-point operations: Unavailable–2; Available–10.

9. Dynamic reconfigurability of instructions: Not Possible – 1; possible – 10.

10. I/O bandwidth for image/video support: Minimum – 4; Provision for large amount of data transfer – 8; Special provision – 10.

11. Support for three levels of computer vision tasks: (i) No support – 1; (ii) reasonable – 5; (iii) Special features – 10.

12. Cost: High cost (>$100,000) – 1; Medium ($10,000 – $30,000) – 5; Low (<$10,000) – 10.

7.1.1 Systems used for the study

The following classes of machines have been chosen for the comparative analysis.

1. Custom Computing Machines (CCMs): Splash 2, Virtual Computer (VC), MORPH.

2. Futuristic FPGA-based CCMs: Based on XC 6200

3. General-purpose parallel processors: SP-2, CM-5 [96].

4. Special-purpose computer vision systems: NETRA [47], IUA [226], VisTA [209].

5. State-of-the-art microprocessors: SuperSPARC, PowerPC

6. Accelerator boards: $i860$-based Alacron

7. Special-purpose processor for computer vision tasks: MVP-80 [214].

8. Special-purpose ASIC: No specific ASIC has been identified, but generic ASIC properties will be used.

7.1.2 Method

To obtain a taxonomy, the following steps are involved:

1. Obtain the 14×12 pattern matrix by assigning suitable values to the 12 features for the 14 platforms chosen for the study.

2. Compute the 14×14 dissimilarity matrix of the platforms from the pattern matrix using a suitable dissimilarity measure. In our case, we have chosen two types of dissimilarity measures: (i) the standard Euclidean distance and (ii) a weighted distance measure computed as follows. The maximum value of an attribute is 10, hence we divide the absolute difference of the attribute value between two patterns by 10 to get a fractional weighted distance with respect to an attribute. A sum of all the attribute distances divided by the number of attributes defines the overall distance between the two patterns. Note that the maximum possible distance between two platforms is 1.0 (totally different patterns) and minimum distance is 0.0 (identical patterns). For example, let

three 12-dimensional feature vectors be

{5 7 6 7 10 5 10 10 1 10 8 10},

{5 10 4 2 2 10 10 10 1 8 1 1},

{10 2 8 10 8 3 7 4 10 6 10 4}.

The distance between the first two patterns =

1/12(0/10+3/10+2/10+5/10+8/10+5/10+0/10+0/100/10+2/10+7/10+9/10)

= 41/120 or 0.341.

Similarly, the distance between patterns 2 and 3 =

1/12(5/10+8/10+4/10+8/10+6/10+7/10+3/10+6/10+9/10+2/10+9/10+3/10)

= 70/120 = 0.58.

3. Apply single-link and complete-link hierarchical clustering algorithms to obtain the taxonomies.

4. Cutting the tree to obtain a partition.

7.2 Results

For the 14 platforms listed in the previous section, the pattern matrix is shown in Table 7.1. On this pattern matrix, principal component analysis and hierarchical clustering have been carried out.

Platform	Id. number	User program.	Ease of use	Comp. speed	Mode of parallelism	Scalability	Design cycle time
Splash 2	1	10	2	8	10	8	3
MORPH	2	10	3	5	10	6	3
VC	3	10	2	8	10	8	3
6200-based	4	10	4	4	10	6	3
CM-5	5	5	7	5	7	9	5
SP-2	6	5	7	6	7	10	5
NETRA	7	5	6	6	4	5	5
IUA	8	5	5	7	4	5	5
VisTA	9	5	4	6	4	6	5
SSPARC	10	5	10	3	2	2	10
PPC	11	5	10	4	2	2	10
i860	12	5	8	5	2	2	8
MVP-80	13	5	4	6	4	4	5
ASIC	14	1	10	10	1	1	1

First six features of the pattern matrix.

Platform	Id. number	PE to PE comm.	FP support	Dynamic reconfig.	I/O Bandwidth	Vision task support	Cost
Splash 2	1	7	4	10	6	10	4
MORPH	2	5	4	10	6	8	2
VC	3	7	4	10	6	10	6
6200-based	4	6	6	10	8	6	4
CM-5	5	10	10	1	10	8	10
SP-2	6	10	10	1	10	8	10
NETRA	7	8	8	1	10	10	10
IUA	8	8	8	1	10	10	10
VisTA	9	8	8	1	10	10	10
SSPARC	10	10	10	1	8	1	1
PPC	11	10	10	1	8	1	1
i860	12	10	10	1	8	3	2
MVP-80	13	8	8	1	8	5	4
ASIC	14	6	10	1	8	10	8

Table 7.1: 14 $\times$ 12 Pattern matrix (shown in two tables).

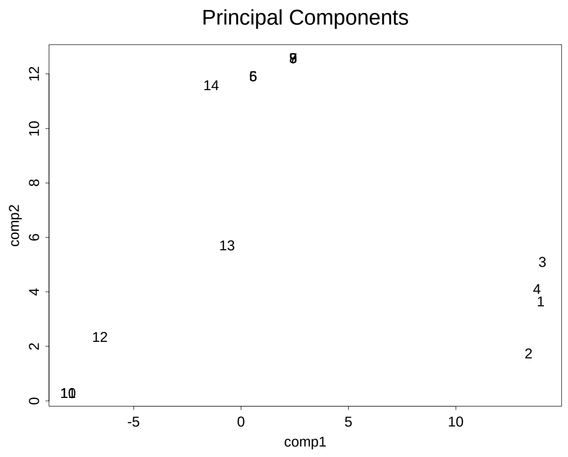


Figure 7.1: Principal component analysis.

7.2.1 Visualization

It is not possible to visualize a 12-dimensional data set. A popular technique for visualization purposes is to project the high dimensional data onto a first few principal axes. The data spread is shown in Figure 7.1 after projecting the 12-dimensional data onto its first two principal axes. The first two principal axes define the axes of maximum variances. The percentage of variance retained by the first two components is 86% as against 53% by the first component alone. When projected to this 2-dimensional plane, some of the patterns may overlap (for example, patterns 10 and 11, and 7 and 8 in Figure 7.1). It can be easily seen that the distinct machine classes have clustered together (e.g., machines {1, 2, 3, and 4}, and {10, 11 and 12}). Using multidimensional scaling, the proximity matrix has been represented in two dimensions in Figure 7.2. Again, we can see several similar patterns grouping together (e.g., {1, 2, 3 and 4}, {5, 6, 7, and 8}, and {9, 10, and 11}). The 'goodness of fit' in terms of a stress value for this case is 0.032.

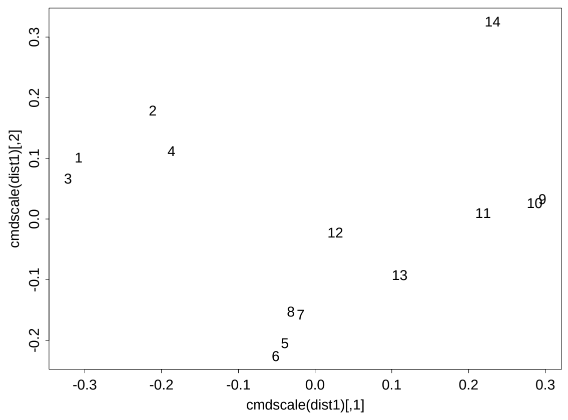


Figure 7.2: Multidimensional scaling using the proposed dissimilarity measure.

7.2.2 Hierarchical clustering

The two proximity matrices obtained using the different distance measures (Euclidean and weighted distance described earlier) are subjected to hierarchical clustering. The output of a hierarchical clustering algorithm is a dendrogram. Using Euclidean distance measure as the dissimilarity measure between the platforms, the single-link and complete-link dendrograms are shown in Figure 7.3 and using the weighted dissimilarity measure are shown in Figure 7.4. The fourteen platforms have been numbered 1 through 14 as shown in the table 7.1. A dendrogram can be used to obtain a partition (clusters) at various levels of dissimilarity. At the highest level of dissimilarity, each pattern is in its own class, and at the lowest level of dissimilarity, all the patterns are clustered into one class. By cutting the dendrogram at a suitable level of dissimilarity, various clusterings (partitions) can be obtained. For example, if we cut the dendrogram shown in Figure 7.3(a) at a dissimilarity value of approximately 8 units, we get four clusters. Similarly, cutting the dendrogram at a level of 12 units, we get only two clusters.

Figure	Level	Classes and the members
Figure 7.3(a)	5	(1 2 3 4) (5 6) (7 8 9) (10 11 12) (13) (14)
Figure 7.3(b)	6	(1 2 3 4) (5 6) (7 8 9) (10 11 12) (13) (14)
Figure 7.4(a)	0.16	(1 2 3 4) (5 6 7 8 12) (9 10 11) (13) (14)
Figure 7.4(b)	0.3	(1 2 3 4) (5 6 7 8 12) (9 10 11) (13) (14)

Table 7.2: Analysis of the dendrograms.

Table 7.2 shows the suggested dissimilarity levels of cutting the dendrogram for the four cases and the cluster membership. Note that the levels can be different and still give rise to the same clusters. There are five distinct groups of the 14 different platforms visible in the hierarchy. The major groups are custom computing machines consisting of patterns numbered 1, 2, 3, and 4; general-purpose parallel machines consisting of patterns 5 and 6; special-purpose parallel machine for vision consisting of patterns 7, 8 and 9; general-purpose uniprocessors consisting of patterns 10, 11 and 12; and special purpose processors such as MVP-80 and ASICs. However, some group members have changed their membership when using the weighted dissimilarity measure. For example, uniprocessor *i*860 joined the parallel processor group. The major groups remain unchanged in the four dendrograms. In a sense, this demonstrates that the feature values assigned to these machines are distinctive enough to form the separate groups.

7.3 Discussion

Using the proposed approach a new hardware platform can be examined for its class membership by assigning appropriate values to the twelve features and building a new dendrogram. This can be easily seen in case of the new hypothetical CCM based on XC 6200 which grouped with the CCM class. A tool can also be built to assign different values to the features and analyze the dendrograms interactively.

7.4 Summary

Using techniques from multidimensional data analysis, we have been able to build a meaningful hierarchy of several computing platforms, including CCMs. As expected, all the CCMs grouped themselves into a single category or cluster. The ASICs and general-purpose processors are quite different from a CCM. Many special-purpose systems are also grouped together. Using these 12-features, a new hardware platform can be assigned to one of these five classes by constructing a dendrogram.

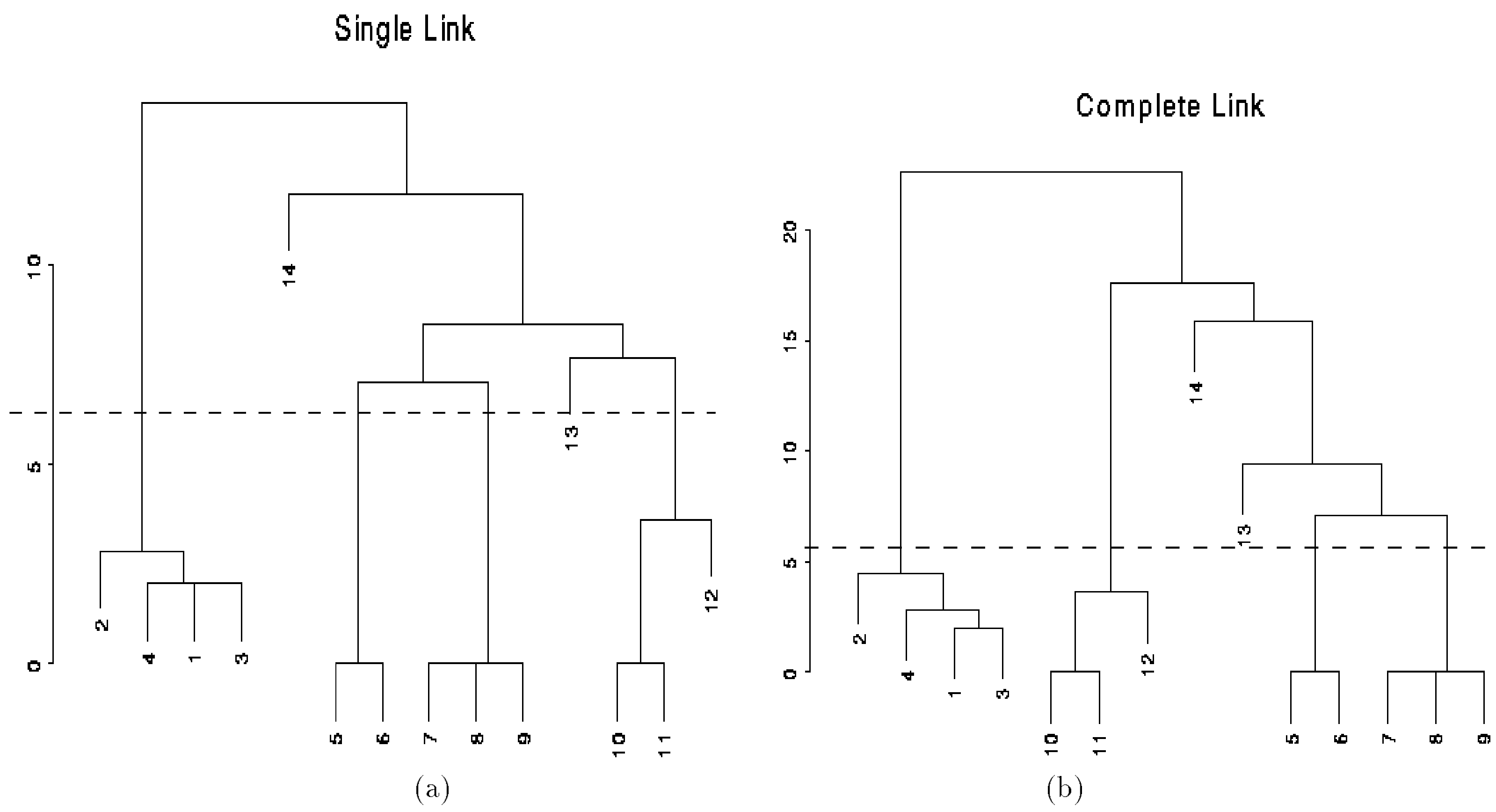
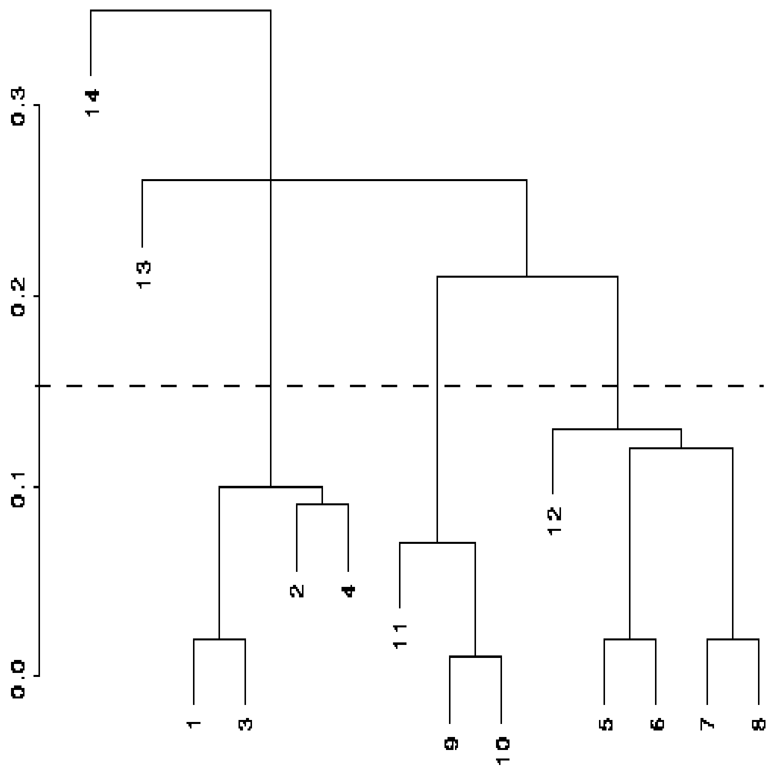
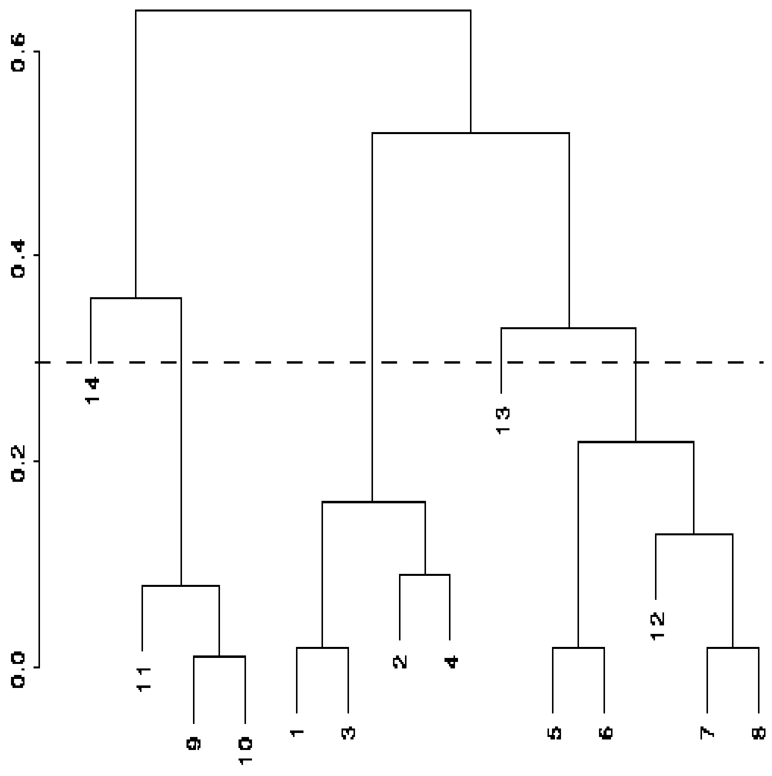


Figure 7.3: Dendrograms showing taxonomy of machines based on Euclidean distance measure. (a) Single link; (b) Complete link.



(a)



(b)

Figure 7.4: Dendrograms showing taxonomy of machines based on proposed distance measure. (a) Single link; (b) Complete link.

Chapter 8

Conclusions and Directions for Future Research

The main goal of this research has been to evaluate suitability of custom computing approach in meeting computational needs of computer vision algorithms. The experiments carried out by way of mapping several representative vision algorithms onto Splash 2 have shown the usefulness of CCMs for computer vision. In addition to achieving high speeds of operation, several other benefits of employing CCMs for computer vision algorithms are listed below:

- Custom computing machines are suitable for all the three levels of computer vision algorithms (low-level, intermediate-level and high-level) by appropriate reconfiguration when required.

- All the stages of a vision algorithm can be implemented on custom computing machines, but an optimal interaction between software and hardware is necessary for best performance.

- The execution speeds achieved on Splash 2 are close to ASIC-level speeds.
- Using the reconfigurability property of the FPGAs, it is easy to reuse the hardware for more than one application at run-time.
- Design revisions are easily supported.
- The PEs can be programmed for systolic, SIMD, MIMD and pipelined mode.
- PE to PE communication patterns can also be programmed.
- Cost performance ratio is significantly low.

Many of the above-mentioned benefits are a result of using FPGAs as the basic compute element. Recent trends in the FPGA technology are directed towards supporting partial reconfigurability and faster reconfigurability. These features will further enhance the utility of CCMs for an easier mapping of complex multi-stage operators.

In mapping the sequential algorithms to a parallel machine, several changes take place in the algorithm to exploit the target architecture characteristics. For example, on a sequential machine, convolution is implemented in a different way than on Splash 2 where we chose a systolic algorithm. In mapping the text segmentation algorithm onto Splash 2, one of the main changes carried out is in the number representation scheme. We have chosen a smaller word length (16-bits) instead of typical 32- or 64-bit representation for floating point numbers. This results in a loss of accuracy.

But, after verifying during simulation that this does not affect the output, it was adopted. In the fingerprint matching algorithm, we have converted computations to a lookup operation. There is no need for computing the common bounding box as the parallel algorithm can tolerate a few extra minutia checks.

It is not true that CCMs are a panacea for all compute-intensive problems. The available technology poses a number of limitations. The limitations of the currently available CCMs are as follows.

- By design, the FPGAs are not meant for complex floating point operations. Complex operators involving multi-stage floating point operations end up with a large number of gates and long delay lines of interconnections. However, recent trends in FPGA technology are encouraging in terms of a larger number of gates per FPGA. This will enable us to synthesize more complex floating point operations at acceptable speeds.

- The FPGA building blocks (CLBs, IOBs and Interconnects) are not fully utilized during the synthesis process. Often, the routing resources get consumed quickly resulting in a low utilization of the CLBs. Better placement tools are being released by the FPGA vendors to overcome this problem. Alternate technologies are being worked out to overcome this problem with the present SRAM-based FPGAs.

- The design process being very complex, the users tend not to accept CCMs as an alternative to high performance computing. Many researchers have attempted to remove this barrier by making the design process transparent through programming in C or C++. But, for best performance, as in a typical parallel processing system, an understanding of the underlying architecture is necessary to optimize the mapping. With low-resource FPGAs, the high language overheads make it difficult to map a complex algorithm. With the availability of large density FPGAs, programming using a high-level language like C or C++ will become feasible.

- Algorithms requiring a large number of gates are difficult to map onto low logic density devices. Vendors are working on FPGAs with 100K gates which will enable mapping of many common applications.

- Often, a partitioning of the problem is done manually. This demands the designer to carry out a detailed analysis of the problem in terms of computation and communication complexity. Researchers are working on automatic techniques for partitioning large designs onto multichip modules.

8.1 Directions for future research

Custom computing machines are currently going through a major evolution. Hence, there are many research issues that need to be addressed. Several research issues are given below.

- The available CCMs lack a user-friendly programming environment. For example, an integrated development environment that can aid a designer to quickly prototype his computer vision algorithm and a more efficient mapping for actual usage will be helpful. A layered interface for different types of users is necessary starting with a total transparent mode to a more detailed and efficient mode.

- The dynamic partial reconfigurability might have a significant impact on design of computer vision systems. This needs to be studied.
- The impact of latest architectural features such as direct memory access by host processor and dedicated bus interface logic may change many computation paradigms. A detailed analysis is needed.
- The formal design approach of hardware-software codesign has not been investigated. This is a promising area of future research.

Appendices

Appendix A

Case Study: Image Segmentation

This appendix contains the C-language code for the text segmentation algorithm explained in Chapter 3, VHDL source code, and the C-language interface code for Splash 2 along with sample *ppr* report and sample make files. It includes the following.

- C-code for image segmentation
- VHDL code for image segmentation
- PPR Summary
- Host C-interface code
- Makefiles

```
/*
  C-program for the simple image segmentation algorithm described in
  Chapter 3. The program needs an image file name, rows, cols and output
  file name. The input file is assumed to be in raw format and the output
  file is a sequence of raw bytes.
*/
#include <stdio.h>
#include <stdlib.h>
#define MAX_ROWS 1024
#define MAX_COLS 1024
#define MAX_WIN 64
#define MAX_MASK 9


unsigned char input_buf[MAX_ROWS][MAX_COLS];
unsigned char output_buf[MAX_ROWS][MAX_COLS];

int mval[MAX_MASK][MAX_MASK];

FILE *input, *output;

int im_width,im_height,window,mask,rows,cols;
long clock();
int debug=1;

main(argc,argv)
int argc;
char *argv[];
{
int i,j;

if (argc<4)
{
 printf("usage is calgo in_image i_row i_col out_image\n");
exit(1);
}
if ((input=fopen(argv[1],"rb"))==NULL)
{
 printf("Error in opening image file %s\n",argv[1]);
 exit(1);
}
im_height = atoi(argv[2]);
im_width= atoi(argv[3]);
if ((im_height>1024) || (im_width>1024))
{
```

```
printf("Error in image size %d %d\n",im_width,im_height);
exit(1);
}
if ((output=fopen(argv[4],"wb"))==NULL)
{
 printf("Error in writing image file %s\n",argv[6]);
 exit(1);
}
window=32;
mask=7;
rows=im_height;
cols=im_width;
for (i=0;i<mask;i++)
 for (j=0;j<mask;j++)
 mval[i][j] = 1;
read_image(input,im_width,im_height);
printf("Time now=%ld\n",clock());
compute_output(rows,cols,window,mask);
printf("Time now=%ld\n",clock());
write_image(output,im_width,im_height);
}
/*end of main*/

read_image(infile,incol,inrow)
FILE *infile;
int incol, inrow;
{
int i;
if (debug)
printf("read_image: Rows=%d,cols=%d\n",inrow,incol);
for (i=0;i<inrow;i++)
fread(&input_buf[i],incol,1,infile);
}
/*end of read_image*/

compute_output(rows,cols,window,mask)
int rows,cols,window,mask;
{
int i,j;
int k,l,k2,sum;
float var1,mean1;

k2 = mask/2;
for (i=window/2;i<rows-window/2;i++)
 for (j=window/2;j<cols-window/2;j++)
```

```
{
     sum = 0;
     var1 = 0;
     for (k=-k2;k<=k2;k++)
      {
      for (l=-k2;l<=k2;l++)
      {
      sum += input_buf[i+k][j+l]*mval[k+k2][l+k2];
      var1 += input_buf[i+k][j+l]*input_buf[i+k][j+l];
      }
      }
     mean1 = sum;
     mean1 = mean1/(mask*mask);
     var1 = var1 - mean1*mean1*mask*mask;/*total elements=msize*msize*/
     var1 = var1/(mask*mask);
     if (var1<250) output_buf[i][j] = 0;
     else if ((var1>=250) && (mean1 < 120)) output_buf[i][j] = 1;
        else output_buf[i][j] = 2;
    }

}
/*end of compute_output*/

write_image(outfile,incol,inrow)
FILE *outfile;
int incol, inrow;
{
int i;
 if (debug)
printf("write_image: Rows=%d,cols=%d\n",inrow,incol);
for (i=0;i<inrow;i++)
fwrite(output_buf[i],incol,1,outfile);
}
/*end of write_image*/
```

```
---------------------------------------------------------------------
--
--     PROGRAM:   X0_XBAR_BROADCAST
--
--        DATE:   24 Apr 95
--
--      AUTHOR:   Nalini Ratha
--
--  DESCRIPTION:  This design broadcasts the data read from memory
--                onto the crossbar. (To X1)
--
---------------------------------------------------------------------

library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
use SPLASH2.ARITHMETIC.all;
library IEEE;
USE IEEE.std_logic_1164.all;


entity Xilinx_Control_Part is
  Generic(
    BD_ID               : Integer := 0;        --  Splash Board
                                               --  ID
    PE_ID               : Integer := 0         --  Processing
                                               --  Element ID
  );
  Port (
    X0_SIMD             : inout DataPath;      --  SIMD Data
                                               --  Bus
    X0_XB_Data          : inout DataPath;      --  Crossbar Data
                                               --  Bus
    X0_Mem_A            : inout MemAddr;       --  Splash Memory
                                               --  Address Bus
    X0_Mem_D            : inout MemData;       --  Splash Memory
                                               --  Data Bus
    X0_Mem_RD_L         : inout RBit3;         --  Splash Memory Read
                                               --  Signal (low-true)
    X0_Mem_WR_L         : inout RBit3;         --  Splash Memory Write
                                               --  Signal (low-true)
    X0_Mem_Disable      : in    Bit;           --  Splash Memory
                                               --  Disable Signal
    X0_GOR_Result_In    : inout RBit3_Vector(1 to XILINX_PER_BOARD);
    X0_GOR_Valid_In     : inout RBit3_Vector(1 to XILINX_PER_BOARD);
    X0_GOR_Result       : out   Bit;             --  Global OR
                                                 --  Result Signal
    X0_GOR_Valid        : out   Bit;             --  Global OR
                                                 --  Valid Signal
    X0_Clk              : in    Bit;             --  Splash System
                                                 --  Clock
    X0_XBar_Set         : out   Bit_Vector(0 to 2);--  Crossbar Set
                                                 --  Signals
    X0_X16_Disable      : out   Bit;             --  X16 Disable
    X0_XBar_Send        : out   Bit;             --  X0 broadcasts
    X0_Int              : out   Bit;             --  Interrupt Signal
    X0_Broadcast_In     : in    Bit;             --  Broadcast Input
    X0_Broadcast_Out    : out   Bit;             --  Broadcast Output
    X0_Reset            : in    Bit;             --  Reset Signal
    X0_HS0              : inout RBit3;           --  Handshake Signal
    X0_HS1              : in    Bit;             --  Handshake Signal
    X0_XBar_EN_L        : out   Bit;             --  Crossbar Enable
                                                 --  (low-true)
    X0_LED              : out   Bit              --  LED Signal
  );
end Xilinx_Control_Part;


architecture X0_XBAR_BROADCAST of Xilinx_Control_Part is

--------------------------------------------------------------
--  Signal Declarations
--------------------------------------------------------------


  SIGNAL Xbar_Out   : Bit_Vector(35 downto 0);
  SIGNAL Data       : Bit_Vector(15 downto 0);
  SIGNAL Address    : Bit_Vector(17 downto 0);
  SIGNAL ONE    : Bit_Vector(17 downto 0);
  SIGNAL out1       : Bit_Vector(7 downto 0);
  SIGNAL out2       : Bit_Vector(7 downto 0);
  SIGNAL out3       : Bit_Vector(7 downto 0);
  SIGNAL result     : Bit_Vector(15 downto 0);
--------------------------------------------------------------
--  Architecture Behavior
--------------------------------------------------------------
```

```
  BEGIN

    --  set X0 to broadcast and disable X16

    X0_XBar_Set <= "000";
    X0_X16_Disable  <= '1';
    X0_xbar_send    <= '1';

    -- Memory read settings

    X0_Mem_RD_L     <= '0';
    X0_Mem_WR_L     <= '1';
    ONE <= "0000000000000000001";
  PROCESS
  BEGIN

    WAIT until X0_Clk'event AND X0_Clk = '1';
    --  connections to I/O pads
    pad_output (X0_Mem_A, Address);
    pad_input (X0_Mem_D, Data);
    Pad_Output (X0_XB_Data, Xbar_Out);

    out1(7 downto 0) <= data(7 downto 0);
    out2(7 downto 0) <= data(7 downto 0);
    out3 <= out1;
    result <= out1 (7 downto 0) * out2 (7 downto 0);
    Xbar_out(7 downto 0) <= out3(7 downto 0);
    Xbar_out(23 downto 8) <= result(15 downto 0);
    Address <= Address+one;
END PROCESS ;
    X0_Int <= '0';
    X0_Broadcast_Out <= '0';
end X0_XBAR_BROADCAST;
```

```
----------------------------------------------------------------------
--
--     PROGRAM: 2-D averaging, stage1
--
--         DATE: 24 Apr. 1995
--
--       AUTHOR: Nalini K. Ratha
--
--  DESCRIPTION: First row for 7x7 mask (1 1 1 1 1 1 1)
--
--
---------------------------------------------------------------------
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
library IEEE;
USE IEEE.std_logic_1164.all;


----------------------------------------------------------------------
--  Splash 2 Simulator v1.5 Xilinx_Processing_Part Entity Declaration
----------------------------------------------------------------------
entity Xilinx_Processing_Part is
  Generic(
    BD_ID              : Integer := 0;        --  Splash Board
                                              --  ID
    PE_ID              : Integer := 0         --  Processing
                                              --  Element ID
);
  Port (
    XP_Left            : inout DataPath;      --  Left Data
                                              --  Bus
    XP_Right           : inout DataPath;      --  Right Data
                                              --  Bus
    XP_Xbar            : inout DataPath;      --  Crossbar Data
                                              --  Bus
    XP_Xbar_EN_L       : out   Bit_Vector(4 downto 0);
                                              --  Crossbar Enable
                                              --  (low-true)
    XP_Clk             : in    Bit;           --  System Clock
    XP_Int             : out   Bit;           --  Interrupt Signal
    XP_Mem_A           : inout MemAddr;       --  Memory Address Bus
    XP_Mem_D            : inout MemData;        --  Memory Data Bus
    XP_Mem_RD_L         : inout RBit3;          --  Memory Read
                                                --  (low- true)
    XP_Mem_WR_L         : inout RBit3;          --  Memory Write Signal
                                                --  (low-true)
    XP_Mem_Disable      : in    Bit;            --  Memory Disable
    XP_Broadcast        : in    Bit;            --  Broadcast Signal
    XP_Reset            : in    Bit;            --  Reset Signal
    XP_HS0              : inout RBit3;          --  Handshake Signal
    XP_HS1              : in    Bit;            --  Handshake Signal
    XP_GOR_Result       : inout RBit3;          --  Global OR Result
    XP_GOR_Valid        : inout RBit3;          --  Global OR Valid
    XP_LED              : out   Bit             --  LED Signal
  );
end Xilinx_Processing_Part;

-----------------------------------------------------------------------
--  Architecture
-----------------------------------------------------------------------

ARCHITECTURE conv_s1_f1 OF Xilinx_Processing_Part IS
 Constant STAGES: integer:= 7;
 Constant REGS: integer:=30;
 type bvarray is array (1 to REGS) of Bit_vector(15 downto 0);
 type pixarray is array (1 to STAGES) of Bit_vector(15 downto 0);
 type sumarray is array (1 to STAGES) of Bit_vector(15 downto 0);
------------------------------------------------------------------------
--  Signal Declarations
------------------------------------------------------------------------

  Signal datain: bvarray; ---sum partial

  SIGNAL left_in : pixarray;
  SIGNAL tsum: sumarray;
  SIGNAL tsum1: sumarray;
  SIGNAL part_sum: sumarray;
  SIGNAL left_sum: Bit_Vector (15 downto 0);
  SIGNAL right_out:  Bit_Vector(35 downto 0);
  SIGNAL input_left: Bit_Vector (35 downto 0);
  SIGNAL add, sub,ofl6,ofl1,ofl2,ofl3,ofl4,ofl5,ofl7: Bit;
-------------------------------------------------------------------------
--  Architecture Behavior
-------------------------------------------------------------------------

  BEGIN
  XP_Xbar_En_L    <= "00000";
  add <= '1';
  sub <= '0';
  PROCESS
  BEGIN

  WAIT until Xp_Clk'event AND XP_Clk = '1';
    Pad_Input  (XP_Xbar, input_left);
    Pad_output (XP_Right, right_out);
    for i in 2 to STAGES loop
     left_in(i)(7 downto 0) <= left_in(i-1)(7 downto 0);
    end loop;
    left_in(1)(7 downto 0) <= input_left(7 downto 0);
    left_sum <= itobv(0,16);
-- partial sums to be computed here using hardmacros
-- PE 1
   tsum1(1) <= tsum(1);
   part_sum(1) <= tsum1(1);
-- PE 2
   tsum1(2) <= tsum(2);
   part_sum(2) <= tsum1(2);
-- PE 3
   tsum1(3) <= tsum(3);
   part_sum(3) <= tsum1(3);
-- PE 4
   tsum1(4) <= tsum(4);
   part_sum(4) <= tsum1(4);
-- PE 5
   tsum1(5) <= tsum(5);
   part_sum(5) <= tsum1(5);
-- PE 6
   tsum1(6) <= tsum(6);
   part_sum(6) <= tsum1(6);
-- PE 7
   tsum1(7) <= tsum(7);
   part_sum(7) <= tsum1(7);

-- Shifters

  for i in 2 to REGS loop
   datain(i) <= datain(i-1);
   end loop;
   datain(1)(15 downto 0) <= part_sum(7);
   right_out(23 downto 8) <= datain(REGS)(15 downto 0);
   right_out(7 downto 0) <= left_in(STAGES)(7 downto 0);
```

```
 END PROCESS;
addsub1:
       adsu16h
                port map(left_in(1),left_sum,add,tsum(1),ofl1);

addsub2:
        adsu16h
port map(left_in(2), part_sum(1),add, tsum(2),ofl2);
addsub3:
adsu16h
port map(left_in(3), part_sum(2),add, tsum(3), ofl3);
addsub4:
adsu16h
port map(left_in(4), part_sum(3),add, tsum(4), ofl4);
addsub5:
adsu16h
port map(part_sum(4),left_in(5),add,tsum(5),ofl5);
addsub6:
adsu16h
port map(part_sum(5),left_in(6),add,tsum(6),ofl6);
addsub7:
adsu16h
port map(part_sum(6),left_in(7),add,tsum(7),ofl7);
END conv_s1_f1;
```

```
-------------------------------------------------------------------
--
--     PROGRAM: 2-D convolution, stage2
--
--        DATE: 24 Apr 1995
--
--      AUTHOR: Nalini K. Ratha
--
--  DESCRIPTION: First row for 7x7 mask (Averaging 1  1 1 1 1 1 1)
--
--
--------------------------------------------------------------------
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
library IEEE;
USE IEEE.std_logic_1164.all;


--------------------------------------------------------------------
--  Architecture of Self Diagnostic
--------------------------------------------------------------------

ARCHITECTURE conv_s2_f1 OF Xilinx_Processing_Part IS
 Constant STAGES: integer:= 7;
 Constant REGS: integer:=30;
 type bvarray is array (1 to REGS) of Bit_vector(15 downto 0);
 type pixarray is array (1 to STAGES) of Bit_vector(15 downto 0);
 type sumarray is array (1 to STAGES) of Bit_vector(15 downto 0);
--------------------------------------------------------------------
--  Signal Declarations
--------------------------------------------------------------------

  Signal datain: bvarray; ---sum partial

  SIGNAL left_in : pixarray;
  SIGNAL tsum: sumarray;
  SIGNAL tsum1: sumarray;
  SIGNAL part_sum: sumarray;
  SIGNAL left_sum: Bit_Vector (15 downto 0);
  SIGNAL right_out:  Bit_Vector(35 downto 0);
  SIGNAL input_left: Bit_Vector (35 downto 0);
  SIGNAL add, sub,ofl6,ofl1,ofl2,ofl3,ofl4,ofl5,ofl7: Bit;
--------------------------------------------------------------------
--  Architecture Behavior
--------------------------------------------------------------------

  BEGIN
  XP_Xbar_En_L    <= "00000";
  add <= '1';
  sub <= '0';
  PROCESS
  BEGIN

  WAIT until Xp_Clk'event AND XP_Clk = '1';
    Pad_Input  (XP_Left, input_left);
    Pad_output (XP_Right, right_out);
    for i in 2 to STAGES loop
     left_in(i)(7 downto 0) <= left_in(i-1)(7 downto 0);
    end loop;
    left_in(1)(7 downto 0) <= input_left(7 downto 0);
    left_sum <= input_left(23 downto 8);
-- partial sums to be computed here using hardmacros
-- PE 1
--   tsum(1) <= left_in(1)+left_sum;
   tsum1(1) <= tsum(1);
   part_sum(1) <= tsum1(1);
-- PE 2
--   tsum(2) <= part_sum(1)+left_in(2);
   tsum1(2) <= tsum(2);
   part_sum(2) <= tsum1(2);
-- PE 3
--   tsum(3) <= part_sum(2)+left_in(3);
   tsum1(3) <= tsum(3);
   part_sum(3) <= tsum1(3);
-- PE 4
--   tsum(4) <= part_sum(3)+left_in(4);
   tsum1(4) <= tsum(4);
   part_sum(4) <= tsum1(4);
-- PE 5
--   tsum(5) <= part_sum(4)+left_in(5);
   tsum1(5) <= tsum(5);
   part_sum(5) <= tsum1(5);
-- PE 6
--   tsum(6) <= part_sum(5)+left_in(6);
   tsum1(6) <= tsum(6);
   part_sum(6) <= tsum1(6);
```

```
-- PE 7
--   tsum(7) <= part_sum(6)+left_in(7);
   tsum1(7) <= tsum(7);
   part_sum(7) <= tsum1(7);

-- Shifters

  for i in 2 to REGS loop
   datain(i) <= datain(i-1);
   end loop;
   datain(1)(15 downto 0) <= part_sum(7);
   right_out(23 downto 8) <= datain(REGS)(15 downto 0);
   right_out(7 downto 0) <= left_in(STAGES)(7 downto 0);
 END PROCESS;
addsub1:
       adsu16h
                port map(left_in(1),left_sum,add,tsum(1),ofl1);

addsub2:
        adsu16h
port map(left_in(2), part_sum(1),add, tsum(2),ofl2);
addsub3:
adsu16h
port map(left_in(3), part_sum(2),add, tsum(3), ofl3);
addsub4:
adsu16h
port map(left_in(4), part_sum(3),add, tsum(4), ofl4);
addsub5:
adsu16h
port map(part_sum(4),left_in(5),add,tsum(5),ofl5);
addsub6:
adsu16h
port map(part_sum(5),left_in(6),add,tsum(6),ofl6);
addsub7:
adsu16h
port map(part_sum(6),left_in(7),add,tsum(7),ofl7);
END conv_s2_f1;
```

```
---------------------------------------------------------------
--
--      PROGRAM: 2-D convolution, stage1
--
--          DATE: 24 April 1995
--
--        AUTHOR: Nalini K. Ratha
--
--  DESCRIPTION: Seventh row for 7x7 mask (1  1 1 1 1 1 1)
--
--
---------------------------------------------------------------
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
library IEEE;
USE IEEE.std_logic_1164.all;


---------------------------------------------------------------
--  Architecture of Self Diagnostic
---------------------------------------------------------------

ARCHITECTURE conv_s7_f1 OF Xilinx_Processing_Part IS
 Constant STAGES: integer:= 7;
 Constant REGS: integer:=30;
 type bvarray is array (1 to REGS) of Bit_vector(15 downto 0);
 type pixarray is array (1 to STAGES) of Bit_vector(15 downto 0);
 type sumarray is array (1 to STAGES) of Bit_vector(15 downto 0);
---------------------------------------------------------------
--  Signal Declarations
---------------------------------------------------------------

  Signal datain: bvarray; ---sum partial

  SIGNAL left_in : pixarray;
  SIGNAL tsum: sumarray;
  SIGNAL tsum1: sumarray;
  SIGNAL part_sum: sumarray;
  SIGNAL left_sum: Bit_Vector (15 downto 0);
  SIGNAL right_out:  Bit_Vector(35 downto 0);
  SIGNAL input_left: Bit_Vector (35 downto 0);
  SIGNAL add, sub,ofl6,ofl1,ofl2,ofl3,ofl4,ofl5,ofl7: Bit;
---------------------------------------------------------------
--  Architecture Behavior
---------------------------------------------------------------

  BEGIN
--  XP_Xbar_En_L    <= "11100";
  XP_Xbar_En_L <= "11111";
  add <= '1';
  sub <= '0';
  PROCESS
  BEGIN

  WAIT until Xp_Clk'event AND XP_Clk = '1';
    Pad_Input  (XP_Left, input_left);
    Pad_output (XP_Xbar, right_out);
    for i in 2 to STAGES loop
     left_in(i)(7 downto 0) <= left_in(i-1)(7 downto 0);
    end loop;
    left_in(1)(7 downto 0) <= input_left(7 downto 0);
    left_sum <= input_left(23 downto 8);
-- partial sums to be computed here using hardmacros
-- PE 1
   tsum1(1) <= tsum(1);
   part_sum(1) <= tsum1(1);
-- PE 2
   tsum1(2) <= tsum(2);
   part_sum(2) <= tsum1(2);
-- PE 3
   tsum1(3) <= tsum(3);
   part_sum(3) <= tsum1(3);
-- PE 4
   tsum1(4) <= tsum(4);
   part_sum(4) <= tsum1(4);
-- PE 5
   tsum1(5) <= tsum(5);
   part_sum(5) <= tsum1(5);
-- PE 6
   tsum1(6) <= tsum(6);
   part_sum(6) <= tsum1(6);
-- PE 7
   tsum1(7) <= tsum(7);
   part_sum(7) <= tsum1(7);

-- Shifters
```

```
  for i in 2 to REGS loop
   datain(i) <= datain(i-1);
   end loop;
   datain(1)(15 downto 0) <= part_sum(7);
   right_out(15 downto 0) <= datain(REGS)(15 downto 0);
   right_out(23 downto 16) <= left_in(STAGES)(7 downto 0);
 END PROCESS;
addsub1:
       adsu16h
                port map(left_in(1),left_sum,add,tsum(1),ofl1);

addsub2:
        adsu16h
port map(left_in(2), part_sum(1),add, tsum(2),ofl2);
addsub3:
adsu16h
port map(left_in(3), part_sum(2),add, tsum(3), ofl3);
addsub4:
adsu16h
port map(left_in(4), part_sum(3),add, tsum(4), ofl4);
addsub5:
adsu16h
port map(part_sum(4),left_in(5),add,tsum(5),ofl5);
addsub6:
adsu16h
port map(part_sum(5),left_in(6),add,tsum(6),ofl6);
addsub7:
adsu16h
port map(part_sum(6),left_in(7),add,tsum(7),ofl7);
END conv_s7_f1;
```

```
-----------------------------------------------------------------
--
--     PROGRAM: 2-D convolution, stage1
--
--        DATE: 24  Apr 1995
--
--      AUTHOR: Nalini K. Ratha
--
--  DESCRIPTION: First row for 7x7 mask (1  1 1 1 1 1 1)
--
--
-----------------------------------------------------------------
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
library IEEE;
USE IEEE.std_logic_1164.all;
-----------------------------------------------------------------
--  Architecture of Self Diagnostic
-----------------------------------------------------------------

ARCHITECTURE conv_s1_f2 OF Xilinx_Processing_Part IS
 Constant STAGES: integer:= 7;
 Constant REGS: integer:=30;
 type bvarray is array (1 to REGS) of Bit_vector(15 downto 0);
 type pixarray is array (1 to STAGES) of Bit_vector(15 downto 0);
 type sumarray is array (1 to STAGES) of Bit_vector(15 downto 0);
-----------------------------------------------------------------
--  Signal Declarations
-----------------------------------------------------------------

  Signal datain: bvarray; ---sum partial

  SIGNAL left_in : pixarray;
  SIGNAL tsum: sumarray;
  SIGNAL tsum1: sumarray;
  SIGNAL part_sum: sumarray;
  SIGNAL left_sum: Bit_Vector (15 downto 0);
  SIGNAL right_out:  Bit_Vector(35 downto 0);
  SIGNAL input_left: Bit_Vector (35 downto 0);
  SIGNAL add, sub,ofl6,ofl1,ofl2,ofl3,ofl4,ofl5,ofl7: Bit;
-----------------------------------------------------------------
```

```
--  Architecture Behavior
------------------------------------------------------------------

  BEGIN
  XP_Xbar_En_L    <= "00000";
  add <= '1';
  sub <= '0';
  PROCESS
  BEGIN

  WAIT until Xp_Clk'event AND XP_Clk = '1';
    Pad_Input  (XP_Xbar, input_left);
    Pad_output (XP_Right, right_out);
    for i in 2 to STAGES loop
     left_in(i)(7 downto 0) <= left_in(i-1)(7 downto 0);
    end loop;
    left_in(1)(7 downto 0) <= input_left(23 downto 16);
-- 10 bit x^2/256 value
--    left_sum <= itobv(0,16);
-- partial sums to be computed here using hardmacros;
-- tsum and tsum1 are to be used accordingly
-- PE 1
   tsum1(1) <= tsum(1);
   part_sum(1) <= tsum1(1);
-- PE 2
   tsum1(2) <= tsum(2);
   part_sum(2) <= tsum1(2);
-- PE 3
   tsum1(3) <= tsum(3);
   part_sum(3) <= tsum1(3);
-- PE 4
   tsum1(4) <= tsum(4);
   part_sum(4) <= tsum1(4);
-- PE 5
   tsum1(5) <= tsum(5);
   part_sum(5) <= tsum1(5);
-- PE 6
   tsum1(6) <= tsum(6);
   part_sum(6) <= tsum1(6);
-- PE 7
   tsum1(7) <= tsum(7);
   part_sum(7) <= tsum1(7);

-- Shifters
```

```
  for i in 2 to REGS loop
   datain(i) <= datain(i-1);
   end loop;
   datain(1)(15 downto 0) <= part_sum(7);
   right_out(23 downto 8) <= datain(REGS)(15 downto 0);
   right_out(7 downto 0) <= left_in(STAGES)(7 downto 0);
 END PROCESS;
addsub1:
       adsu16h
                port map(left_in(1),left_sum,add,tsum(1),ofl1);

addsub2:
        adsu16h
port map(left_in(2), part_sum(1),add, tsum(2),ofl2);
addsub3:
adsu16h
port map(left_in(3), part_sum(2),add, tsum(3), ofl3);
addsub4:
adsu16h
port map(part_sum(3),left_in(4),add,tsum(4),ofl4);
addsub5:
adsu16h
port map(part_sum(4),left_in(5),add,tsum(5),ofl5);
addsub6:
adsu16h
port map(part_sum(5),left_in(6),add,tsum(6),ofl6);
addsub7:
adsu16h
port map(part_sum(6),left_in(7),add,tsum(7),ofl7);
END conv_s1_f2;
```

```
-----------------------------------------------------------------------
--
--     PROGRAM: 2-D convolution, result stage
--
--        DATE: 4/25/95
--
--      AUTHOR: Nalini K. Ratha
--
--  DESCRIPTION: From sigma x and sigma X^2, compute variance and
--               decide the pixel class label
--
--
-----------------------------------------------------------------------
library SPLASH2;
use SPLASH2.TYPES.all;
use SPLASH2.SPLASH2.all;
use SPLASH2.ARITHMETIC.all;
use SPLASH2.COMPONENTS.all;
use SPLASH2.HMACROS.all;
library IEEE;
USE IEEE.std_logic_1164.all;


-----------------------------------------------------------------------
--  Architecture of Self Diagnostic
-----------------------------------------------------------------------

ARCHITECTURE conv_s15 OF Xilinx_Processing_Part IS
-----------------------------------------------------------------------
--  Signal Declarations
-----------------------------------------------------------------------

  SIGNAL Data     : Bit_Vector(15 downto 0);
  SIGNAL Address  : Bit_Vector(17 downto 0);

  SIGNAL right: Bit_Vector (35 downto 0);
  SIGNAL left: Bit_Vector (35 downto 0);
  SIGNAL input_left: Bit_Vector (35 downto 0);
  SIGNAL p1: Bit_Vector (7 downto 0);
  SIGNAL p2: Bit_vector (15 downto 0);
  SIGNAL mean_sum: Bit_vector(15 downto 0);
  SIGNAL mean_sumold, mean_sumold1: Bit_vector(15 downto 0);
  SIGNAL mean_sqr: Bit_vector(25 downto 0);
  SIGNAL x2_sum,x2_sum_old: Bit_vector(15 downto 0);
  SIGNAL x2_sum_old1: Bit_vector(25 downto 0);
  SIGNAL variance:  Bit_vector (25 downto 0);
  SIGNAL const1,const2: Bit_Vector (15 downto 0);
  SIGNAL one,mean1: Bit_vector(15 downto 0);
  SIGNAL temp_mean,mean: Bit_Vector(9 downto 0);
  SIGNAL tmp_mean,tmean: Bit_vector(9 downto 0);
------------------------------------------------------------------------
--  Architecture Behavior
------------------------------------------------------------------------

  BEGIN
  const1 <= "0010111111011010";-- 250*49 (2FDA)
  const2 <= "0001011010000000";-- 120*49 (1680)
  one <= "0000000000000001";
  XP_Xbar_En_L    <= "00000";
  xp_mem_rd_l <= '1';
  xp_mem_wr_l <= '0';

  PROCESS
  BEGIN

  WAIT until Xp_Clk'event AND XP_Clk = '1';
  Pad_Input  (XP_Left, input_left);
  Pad_Input  (XP_Xbar, left);
  pad_output (XP_RIGHT, right);
  Pad_output(xp_mem_a, address);
  Pad_output(xp_mem_d, data);
  p1 (7 downto 0) <= left(23 downto 16);
  mean_sum (15 downto 0) <= left(15 downto 0);
  temp_mean (9 downto 0) <= left(15 downto 6); -- mean_sum/64
  tmp_mean (7 downto 0) <= left(15 downto 8); -- mean_sum/256
  tmean (5 downto 0) <= left (15 downto 10); --mean_sum/1024
  mean <= temp_mean+tmp_mean+tmean; -- mean_sum/49
  mean1 <= mean_sum;
  mean_sumold <= mean1;
  mean_sumold1 <= mean_sumold;
  mean_sqr (25 downto 0) <= mean * mean1;
  right(7 downto 0) <= p1(7 downto 0);
  right(15 downto 8) <= p2(7 downto 0);
  right(25 downto 16) <= mean(9 downto 0);
  p2 (7 downto 0) <= input_left(7 downto 0);
  x2_sum_old1(25 downto 24) <= one(9 downto 8);
  x2_sum_old1(23 downto 8) <=  x2_sum_old (15 downto 0);
  x2_sum_old1(7 downto 0) <= one(7 downto 0);
  x2_sum_old <= x2_sum ;
  x2_sum(15 downto 0) <= input_left(23 downto 8);
  if (x2_sum_old1<mean_sqr) then -- resolution loss error
   variance<=itobv(0,26);
  else
  variance <= x2_sum_old1  - mean_sqr;
  end if;
 if (variance<const1) then
  data(1 downto 0) <= "00";
 elsif (mean_sumold1<const2) then
  data(1 downto 0) <= "10";
 else data(1 downto 0) <= "01";
end if;
   Address <= Address + one;
 END PROCESS;
END conv_s15;
```

```
PPR RESULTS FOR DESIGN PE_15                              Page   i

Table of Contents
-----------------
```

```
Design Statistics and Device Utilization
----------------------------------------



Partitioned Design Utilization Using Part 4010PG191-6


                                 No. Used   Max Available   % Used
  ----------------------------    -------   -------------   ------
  Occupied CLBs                       395         400          98%
  Packed CLBs                         249         400          62%
  ----------------------------    -------   -------------   ------
  Bonded I/O Pins:                    128         160          80%
  F and G Function Generators:        498         800          62%
  H  Function Generators:             72          400          18%
  CLB Flip Flops:                     246         800          30%
  IOB Input Flip Flops:               48          160          30%
  IOB Output Flip Flops:              46          160          28%
  Memory Write Controls:              0           400          0%
  3-State Buffers:                    0           880          0%
  3-State Half Longlines:             0           80           0%
  Edge Decode Inputs:                 0           240          0%
  Edge Decode Half Longlines:         0           32           0%


Routing Summary

  Number of unrouted connections: 0
```

```
PPR Parameters

  Design          = pe_15.xtf
  Parttype        = 4010PG191-6
  Guide_cell      =
  Seed            = 12345
  Estimate        = FALSE
  Complete        = TRUE
  Placer_effort   = 2
  Router_effort   = 2
  Path_timing     = TRUE
  Stop_on_miss    = FALSE
  DC2S            = none
  DP2S            = none
  DC2P            = none
  DP2P            = none
  Guide_only      = FALSE
  Ignore_maps     = FALSE
  Ignore_rlocs    = FALSE
  Outfile         = <design name>


PPR RESULTS FOR DESIGN PE_15                             Page   2

CPU Times

  Partition:                      00:00:44
  Placement:                      00:25:31
  Routing:                        00:21:04
  Total:                          00:47:40


PPR RESULTS FOR DESIGN PE_15                             Page   3

Xact Performance Summary
----------------------------------------

    Deadline  Actual(*)  Specification
    --------  ---------  -------------
(*)   40.0ns  164.6ns    TS0=clock to setup:
      <auto>   25.0ns    <default> pad to setup
      40.0ns   25.0ns    TS48=pad to setup:XP_Left_21_
      40.0ns   25.0ns    TS47=pad to setup:XP_Left_13_
```

```
40.0ns   25.0ns    TS46=pad to setup:XP_Left_9_
40.0ns   25.0ns    TS45=pad to setup:XP_Left_19_
40.0ns   25.0ns    TS44=pad to setup:XP_Xbar_23_
40.0ns   25.0ns    TS43=pad to setup:XP_Xbar_15_
40.0ns   25.0ns    TS42=pad to setup:XP_Left_3_
40.0ns   25.0ns    TS41=pad to setup:XP_Xbar_4_
40.0ns   25.0ns    TS40=pad to setup:XP_Left_20_
40.0ns   25.0ns    TS39=pad to setup:XP_Left_12_
40.0ns   25.0ns    TS38=pad to setup:XP_Left_18_
40.0ns   25.0ns    TS37=pad to setup:XP_Xbar_22_
40.0ns   25.0ns    TS36=pad to setup:XP_Xbar_14_
40.0ns   25.0ns    TS35=pad to setup:XP_Left_4_
40.0ns   25.0ns    TS34=pad to setup:XP_Xbar_3_
40.0ns   25.0ns    TS33=pad to setup:XP_Left_11_
40.0ns   25.0ns    TS32=pad to setup:XP_Xbar_9_
40.0ns   25.0ns    TS31=pad to setup:XP_Left_17_
40.0ns   25.0ns    TS30=pad to setup:XP_Xbar_21_
40.0ns   25.0ns    TS29=pad to setup:XP_Xbar_13_
40.0ns   25.0ns    TS28=pad to setup:XP_Left_5_
40.0ns   25.0ns    TS27=pad to setup:XP_Xbar_2_
40.0ns   25.0ns    TS26=pad to setup:XP_Xbar_19_
40.0ns   25.0ns    TS25=pad to setup:XP_Left_10_
40.0ns   25.0ns    TS24=pad to setup:XP_Xbar_8_
40.0ns   25.0ns    TS23=pad to setup:XP_Left_16_
40.0ns   25.0ns    TS22=pad to setup:XP_Xbar_20_
40.0ns   25.0ns    TS21=pad to setup:XP_Xbar_12_
40.0ns   25.0ns    TS20=pad to setup:XP_Left_6_
40.0ns   25.0ns    TS19=pad to setup:XP_Xbar_1_
40.0ns   25.0ns    TS18=pad to setup:XP_Xbar_18_
40.0ns   25.0ns    TS17=pad to setup:XP_Left_0_
40.0ns   25.0ns    TS16=pad to setup:XP_Xbar_7_
40.0ns   25.0ns    TS15=pad to setup:XP_Left_23_
40.0ns   25.0ns    TS14=pad to setup:XP_Left_15_
40.0ns   25.0ns    TS13=pad to setup:XP_Xbar_11_
40.0ns   25.0ns    TS12=pad to setup:XP_Left_7_
40.0ns   25.0ns    TS11=pad to setup:XP_Xbar_0_
40.0ns   25.0ns    TS10=pad to setup:XP_Xbar_17_
40.0ns   25.0ns    TS9=pad to setup:XP_Left_1_
40.0ns   25.0ns    TS8=pad to setup:XP_Xbar_6_
40.0ns   25.0ns    TS7=pad to setup:XP_Left_22_
40.0ns   25.0ns    TS6=pad to setup:XP_Left_14_
40.0ns   25.0ns    TS5=pad to setup:XP_Xbar_10_
40.0ns   25.0ns    TS4=pad to setup:XP_Left_8_
40.0ns   25.0ns    TS3=pad to setup:XP_Xbar_16_
40.0ns   25.0ns    TS2=pad to setup:XP_Left_2_
```

```
      40.0ns   25.0ns    TS1=pad to setup:XP_Xbar_5_
      <auto>    7.5ns    DEFAULT_FROM_FFS_TO_PADS=FROM:ffs:TO:pads

(*) Note: the actual path delays computed by PPR indicate that 1 of
49 timing specifications you provided was not met.  To confirm this
result, please use the -FailedSpec and/or -TSMaxpaths options of the



------------Reprt has been truncated here for brievity -------------
```

```
/*
  Sample host interface program for Splash 2. This C-code uses the
  raw file generated by t2 for the simple page layout segmentation
  algorithm explained in Chapter 3. The makefile to compile this file
  given separately.
*/
#include <stdio.h>
#include <string.h>
#include "csplash.h"

SPLASH *splash;
Stream *stream;
stepfile *in,*out;
int simd_data[1200], tag[1200];

long clock();
main(argc,argv)
int argc;
char *argv[];
{
int unit=0, board = 0;
int i,j;
FILE *fp, *fp1,*fp2;
int sim_val,sim_tag;
int start,skip,elem_row,elem_col;
int rbus,rbustag,cycles;

char *infile="image.dat",
     *outfile="output.conv",
     *rawfile="mult.raw",
     memfile [50],
     *xbarfile="broadcast.bar";
char fname[50],count;

EnvInit();
SetMessageLevel(1);
if ((splash = OpenAndInit(unit)) == NULL) exit();
if (LoadRaw(splash,board,rawfile) !=0) exit();
if (ConfigArray(splash,board) !=0) exit();
ConfigXBar(splash,0,xbarfile);
ClearMem(splash,0,0);
if (argc>=2)
strcpy(fname,argv[1]);
else
strcpy(fname,infile);
printf("%d %s %s\n",argc,infile,argv[1]);
if (argc>=3)
start=atoi(argv[2]);
else start=0;
skip=0;
if (argc>=4)
elem_row=atoi(argv[3]);
else elem_row=32;
elem_col=32;
if (argc>=5)
cycles=atoi(argv[4]);
else cycles=0;
LoadMem(splash,0,0,fname);
ClearMem(splash,0,15);
printf("Clock= %ld\n",clock());
Step(splash,cycles);
DumpMem(splash,board,15,300,100);*/
j=38*38;
fp2 = fopen("output.conv","w");
count = 0;
for (i=start;i<start+j;i++)
{
int result,result1;
if (i & 1)
{
result = CEMEM(splash,0,15,i);
result1 = result & 0xffff0000;
result1 = result1 >> 16;
result1 = result1 & 0xffff;
fprintf(fp2,"%d %d ",result & 0xffff, result1);
count +=2;
}
if (count==elem_row)
{
fprintf(fp2,"\n");
i = i+skip;
count = 0;
}
}
fprintf(fp2,"\n");
}
/*end of main*/
```

```
SIMULATOR=${SPLASH2}/simulator
TYPES=${SPLASH2}/lib/sim/SPLASH2.sim ${SPLASH2}/lib/sim/TYPES.sim
MODULES=${SIMULATOR}/s2board/SPLASH2_BOARD.sim \
${SIMULATOR}/interface/INTERFACE_BOARD.sim

LIB=DEFAULT

TOP.sim: ${SPLASH2} ${MODULES} SPLASH_SYSTEM.sim pe_0.vhd \
                pe_1.vhd pe_2.vhd pe_7.vhd pe_8.vhd pe_9.vhd \
                pe_14.vhd pe_15.vhd
vhdlan -nc pe_0.vhd
vhdlan -nc pe_1.vhd
vhdlan -nc pe_2.vhd
vhdlan -nc pe_7.vhd
vhdlan -nc pe_8.vhd
vhdlan -nc pe_9.vhd
vhdlan -nc pe_14.vhd
vhdlan -nc pe_15.vhd
vhdlan -w ${LIB} -nc config

SPLASH_SYSTEM.sim: ${SPLASH2} ${MODULES} system.vhd
vhdlan -w ${LIB} -nc system

gcc -I/home/pixel/l46/splash/include -g -w -fvolatile -o cver \
             cver.c /home/pixel/l46/splash/lib/libsplash.a -lm
```

Appendix B

Image Segmentation: Mask Values

```
-------------------------------------------------------
-0.4037 +0.0000 +0.0000 -0.6424 +0.0000 +0.0000 +3.6928
+0.0000 -0.3092 +0.0000 -0.0715 +0.0000 -0.3984 +0.0000
+0.0000 +0.0000 -0.1395 +0.0650 -0.1064 +0.0000 +0.0000
-0.2518 +0.0678 -0.0660 -0.2237 +0.0398 -0.0633 -0.1045
+0.0000 +0.0000 -0.4116 +0.1355 -0.3202 +0.0000 +0.0000
+0.0000 -0.3038 +0.0000 +0.0739 +0.0000 -0.1231 +0.0000
-0.1239 +0.0000 +0.0000 -0.3068 +0.0000 +0.0000 -0.2400
-------------------------------------------------------

-------------------------------------------------------
+0.0039 +0.0000 +0.0000 -0.0320 +0.0000 +0.0000 +0.1824
+0.0000 -0.0688 +0.0000 +0.0578 +0.0000 +0.0006 +0.0000
+0.0000 +0.0000 +0.0536 -0.1939 +0.0223 +0.0000 +0.0000
-0.0557 -0.0223 -0.0572 +0.6117 -0.0740 -0.1647 -0.1797
+0.0000 +0.0000 -0.0678 -7.2124 +0.2979 +0.0000 +0.0000
+0.0000 -0.0437 +0.0000 +7.1524 +0.0000 +0.0161 +0.0000
-0.0446 +0.0000 +0.0000 -0.8858 +0.0000 +0.0000 +0.1908
-------------------------------------------------------

-------------------------------------------------------
-0.1411 +0.0000 +0.0000 -0.0023 +0.0000 +0.0000 +0.0987
+0.0000 +0.1215 +0.0000 +0.1034 +0.0000 -0.7932 +0.0000
+0.0000 +0.0000 +0.1215 -1.1971 +2.3840 +0.0000 +0.0000
+0.0208 +0.4183 +0.0501 +1.0756 -2.5318 +1.1096 -0.2099
+0.0000 +0.0000 -0.4535 +0.3165 +0.1320 +0.0000 +0.0000
+0.0000 +0.0461 +0.0000 +0.1203 +0.0000 +0.2167 +0.0000
```

```
+0.1670 +0.0000 +0.0000 -0.0866 +0.0000 +0.0000 +0.0467
-------------------------------------------------------

-------------------------------------------------------
+0.2002 +0.0000 +0.0000 -0.4906 +0.0000 +0.0000 +0.3070
+0.0000 +0.0098 +0.0000 -0.0600 +0.0000 -0.1655 +0.0000
+0.0000 +0.0000 +0.0808 +0.1343 -0.1308 +0.0000 +0.0000
-0.4092 -0.0506 +0.0674 +0.3235 +0.0860 -0.0235 -0.2146
+0.0000 +0.0000 -0.1318 +0.1686 +0.3258 +0.0000 +0.0000
+0.0000 -0.4292 +0.0000 -0.0216 +0.0000 +0.1819 +0.0000
+0.1806 +0.0000 +0.0000 -0.3611 +0.0000 +0.0000 +0.2897
-------------------------------------------------------

-------------------------------------------------------
+0.4989 +0.0000 +0.0000 +0.6280 +0.0000 +0.0000 +0.5149
+0.0000 -0.9851 +0.0000 +0.4785 +0.0000 -0.7685 +0.0000
+0.0000 +0.0000 +0.3639 -0.3329 +0.4102 +0.0000 +0.0000
-0.8509 +1.4797 -0.2194 -0.9151 +0.8395 -0.0133 -0.1508
+0.0000 +0.0000 +1.4824 -1.1298 +0.0649 +0.0000 +0.0000
+0.0000 -0.5836 +0.0000 +1.1574 +0.0000 +0.1941 +0.0000
+0.2669 +0.0000 +0.0000 -0.6603 +0.0000 +0.0000 -0.1207
-------------------------------------------------------

-------------------------------------------------------
+0.4283 +0.0000 +0.0000 +0.0884 +0.0000 +0.0000 -0.2381
+0.0000 -0.8415 +0.0000 -0.1786 +0.0000 +0.5978 +0.0000
+0.0000 +0.0000 +0.7729 +0.5108 -0.3648 +0.0000 +0.0000
+0.8309 -1.2629 +0.0337 +0.0773 -0.6322 +0.8551 -0.3260
+0.0000 +0.0000 +0.7915 +0.0016 +0.0375 +0.0000 +0.0000
+0.0000 -0.8121 +0.0000 +0.0575 +0.0000 +0.4422 +0.0000
+0.5682 +0.0000 +0.0000 -0.2125 +0.0000 +0.0000 -0.2019
-------------------------------------------------------

-------------------------------------------------------
+0.2880 +0.0000 +0.0000 +0.4107 +0.0000 +0.0000 +0.2070
+0.0000 +0.0933 +0.0000 +0.3882 +0.0000 +0.5268 +0.0000
+0.0000 +0.0000 +0.0100 +0.0745 -3.4039 +0.0000 +0.0000
+0.2192 -0.0740 -0.0623 +0.2686 +0.3411 +0.4821 +0.4482
+0.0000 +0.0000 +0.1331 +0.1360 -0.0323 +0.0000 +0.0000
+0.0000 +0.2039 +0.0000 +0.0074 +0.0000 +0.2564 +0.0000
+0.6176 +0.0000 +0.0000 -0.0149 +0.0000 +0.0000 +0.2959
-------------------------------------------------------

-------------------------------------------------------
-0.2386 +0.0000 +0.0000 +1.4010 +0.0000 +0.0000 -0.2198
```

```
+0.0000 +0.1479 +0.0000 -2.9643 +0.0000 +0.0663 +0.0000
+0.0000 +0.0000 +0.3468 +3.0102 +0.1782 +0.0000 +0.0000
+0.0428 +0.0788 -0.2656 -1.3315 -0.0382 +0.1955 -0.0605
+0.0000 +0.0000 +0.1742 +0.3418 +0.3033 +0.0000 +0.0000
+0.0000 -0.0913 +0.0000 -0.0757 +0.0000 -0.1624 +0.0000
-0.0864 +0.0000 +0.0000 -0.0982 +0.0000 +0.0000 +0.3328
-------------------------------------------------------

-------------------------------------------------------
-0.2570 +0.0000 +0.0000 -0.0499 +0.0000 +0.0000 -0.2175
+0.0000 -0.2593 +0.0000 +0.0267 +0.0000 -0.1406 +0.0000
+0.0000 +0.0000 -0.0093 -0.0185 +0.0634 +0.0000 +0.0000
+0.0177 +0.0698 +0.0666 -0.3496 -0.5230 -0.3649 +0.0018
+0.0000 +0.0000 +0.0076 -0.1180 +3.7845 +0.0000 +0.0000
+0.0000 -0.0723 +0.0000 -0.4575 +0.0000 -0.2661 +0.0000
-0.4318 +0.0000 +0.0000 -0.2779 +0.0000 +0.0000 -0.1841
-------------------------------------------------------

-------------------------------------------------------
-0.3345 +0.0000 +0.0000 -0.1403 +0.0000 +0.0000 -0.2799
+0.0000 -0.1893 +0.0000 -0.0249 +0.0000 +0.0716 +0.0000
+0.0000 +0.0000 -0.2388 +0.0684 -0.2212 +0.0000 +0.0000
-0.2186 -0.0942 +0.2070 +0.0610 -0.0031 -0.4385 +0.0724
+0.0000 +0.0000 -0.0374 +0.0349 +0.1006 +0.0000 +0.0000
+0.0000 -0.3328 +0.0000 +0.0574 +0.0000 -0.8856 +0.0000
-0.1889 +0.0000 +0.0000 -0.1450 +0.0000 +0.0000 +2.1866
-------------------------------------------------------

-------------------------------------------------------
-0.0328 +0.0000 +0.0000 +0.3448 +0.0000 +0.0000 +0.1454
+0.0000 +0.2706 +0.0000 +0.3044 +0.0000 -0.1342 +0.0000
+0.0000 +0.0000 +0.3989 +0.0637 -0.0223 +0.0000 +0.0000
+0.2164 -0.0023 -0.0607 -0.4065 -0.1901 +0.0899 +0.4951
+0.0000 +0.0000 -0.4638 -0.3515 +0.5764 +0.0000 +0.0000
+0.0000 -1.0439 +0.0000 -0.0079 +0.0000 +0.5172 +0.0000
-1.0602 +0.0000 +0.0000 +0.1911 +0.0000 +0.0000 +0.0711
-------------------------------------------------------

-------------------------------------------------------
+0.0377 +0.0000 +0.0000 +0.0855 +0.0000 +0.0000 +0.0889
+0.0000 +0.2255 +0.0000 -0.4636 +0.0000 +0.1686 +0.0000
+0.0000 +0.0000 -1.7636 +1.9047 +0.0271 +0.0000 +0.0000
+0.2639 -0.0649 +2.7799 -2.1575 +0.2629 +0.1183 -0.0920
+0.0000 +0.0000 -0.8930 +0.4933 -0.0554 +0.0000 +0.0000
+0.0000 +0.1137 +0.0000 +0.0258 +0.0000 +0.0617 +0.0000
```

```
-0.1263 +0.0000 +0.0000 +0.0438 +0.0000 +0.0000 -0.0899
-------------------------------------------------------

-------------------------------------------------------
-3.7348 +0.0000 +0.0000 +0.4334 +0.0000 +0.0000 +0.5144
+0.0000 +0.2129 +0.0000 -0.0559 +0.0000 +0.1578 +0.0000
+0.0000 +0.0000 +0.0425 +0.0183 +0.0024 +0.0000 +0.0000
+0.1787 -0.1240 +0.1523 -0.0617 +0.2146 -0.0124 -0.1130
+0.0000 +0.0000 +0.1562 -0.0151 +0.0077 +0.0000 +0.0000
+0.0000 +0.0308 +0.0000 +0.0923 +0.0000 +0.3016 +0.0000
+0.2778 +0.0000 +0.0000 +0.1891 +0.0000 +0.0000 +0.0870
-------------------------------------------------------

-------------------------------------------------------
+0.1401 +0.0000 +0.0000 -0.1543 +0.0000 +0.0000 +0.0080
+0.0000 -0.4049 +0.0000 +0.2715 +0.0000 -0.2464 +0.0000
+0.0000 +0.0000 -0.4327 +0.8048 -0.4185 +0.0000 +0.0000
-0.5282 +1.4128 -1.3855 -0.0496 -0.7359 +1.4604 -1.0166
+0.0000 +0.0000 -0.5496 +1.1765 -0.2960 +0.0000 +0.0000
+0.0000 -0.2833 +0.0000 -0.0613 +0.0000 -0.0374 +0.0000
-0.0375 +0.0000 +0.0000 +0.4585 +0.0000 +0.0000 -0.2479
-------------------------------------------------------

-------------------------------------------------------
+0.0938 +0.0000 +0.0000 -3.2740 +0.0000 +0.0000 +0.1003
+0.0000 +0.0435 +0.0000 +0.5533 +0.0000 +0.4075 +0.0000
+0.0000 +0.0000 +0.3495 -0.2123 +0.2724 +0.0000 +0.0000
+0.0785 -0.0505 +0.1277 -0.2660 -0.2305 -0.0846 -0.0698
+0.0000 +0.0000 +0.2282 -0.0399 +0.0333 +0.0000 +0.0000
+0.0000 -0.1440 +0.0000 -0.1574 +0.0000 +0.2948 +0.0000
+0.1218 +0.0000 +0.0000 -0.0462 +0.0000 +0.0000 +0.1396
-------------------------------------------------------

-------------------------------------------------------
+0.2156 +0.0000 +0.0000 +0.4175 +0.0000 +0.0000 +0.4924
+0.0000 +0.1130 +0.0000 +0.1194 +0.0000 +0.4330 +0.0000
+0.0000 +0.0000 -4.7128 +0.0338 -0.0592 +0.0000 +0.0000
+0.1241 +0.3992 +0.4333 -0.4168 -0.1302 +0.3929 +0.4344
+0.0000 +0.0000 +0.3009 +0.0823 +0.3670 +0.0000 +0.0000
+0.0000 +0.3736 +0.0000 +0.1941 +0.0000 +0.1082 +0.0000
+0.0647 +0.0000 +0.0000 +0.4815 +0.0000 +0.0000 +0.2532
-------------------------------------------------------

-------------------------------------------------------
+0.4460 +0.0000 +0.0000 -0.2423 +0.0000 +0.0000 +0.4429
```

```
+0.0000 +0.0752 +0.0000 +0.0780 +0.0000 -0.0350 +0.0000
+0.0000 +0.0000 +0.0135 -0.0122 +0.0904 +0.0000 +0.0000
+0.0124 +0.4068 +0.3370 +0.2619 +0.2102 -0.0618 +0.1517
+0.0000 +0.0000 -0.0218 -0.0195 +0.1458 +0.0000 +0.0000
+0.0000 +0.1705 +0.0000 +0.3028 +0.0000 +0.0082 +0.0000
-4.1156 +0.0000 +0.0000 +0.1896 +0.0000 +0.0000 +0.4543
-------------------------------------------------------

-------------------------------------------------------
-0.0168 +0.0000 +0.0000 +0.0015 +0.0000 +0.0000 -0.1785
+0.0000 +0.1510 +0.0000 -0.1896 +0.0000 +0.0080 +0.0000
+0.0000 +0.0000 -0.1501 -0.0737 -0.2777 +0.0000 +0.0000
+0.2540 -0.1867 +0.0562 -0.0182 -0.0118 -0.1510 -0.4315
+0.0000 +0.0000 -0.0401 +0.0279 -0.5681 +0.0000 +0.0000
+0.0000 -0.1118 +0.0000 +0.0775 +0.0000 +2.8128 +0.0000
-0.3963 +0.0000 +0.0000 -0.2710 +0.0000 +0.0000 -0.8050
-------------------------------------------------------

-------------------------------------------------------
-0.3502 +0.0000 +0.0000 -0.2298 +0.0000 +0.0000 -0.4247
+0.0000 -0.2707 +0.0000 -0.0388 +0.0000 -0.2208 +0.0000
+0.0000 +0.0000 +0.0472 -0.1824 -0.0578 +0.0000 +0.0000
-0.1394 +0.1580 -0.1588 +0.3303 +0.0160 +0.1176 +0.1681
+0.0000 +0.0000 +0.2153 +0.3370 +0.5260 +0.0000 +0.0000
+0.0000 +0.3873 +0.0000 +0.1582 +0.0000 +0.2344 +0.0000
+0.5111 +0.0000 +0.0000 +0.7181 +0.0000 +0.0000 +0.7858
-------------------------------------------------------

-------------------------------------------------------
+0.0990 +0.0000 +0.0000 +0.2146 +0.0000 +0.0000 +0.0298
+0.0000 +0.1305 +0.0000 -0.1850 +0.0000 -0.0289 +0.0000
+0.0000 +0.0000 +0.2812 +0.1472 -0.0807 +0.0000 +0.0000
-0.0242 +0.0396 +0.1994 -1.7795 +2.1543 -0.6225 +0.2000
+0.0000 +0.0000 -0.1857 +2.4074 -2.1005 +0.0000 +0.0000
+0.0000 -0.0234 +0.0000 -0.4166 +0.0000 +0.6271 +0.0000
+0.1490 +0.0000 +0.0000 +0.3661 +0.0000 +0.0000 -0.0992
-------------------------------------------------------
```

Appendix C

Image Segementation: Neural Network Weights

```
-0.295226 -0.374383 0.028905 -0.009937 -0.396507 0.158501 -0.004866
0.592543 0.136896 0.337228 0.461692 -0.395736 -0.362749 -0.746774
0.673269 0.352101 -0.054250 -0.739054 0.071975 0.530214 0.701421

0.114629 0.365834 -0.778410 0.735883 -0.972597 0.489641 0.424292
0.428273 0.748186 -0.015143 -0.558906 -0.924168 0.142620 0.801619
0.146686 -0.237871 -0.041541 1.001921 0.096374 0.125551 0.204167

-1.371050 3.215525 2.778004 2.079839 0.531245 -0.038866 -0.439274
-2.801049 1.451849 -2.737679 -3.560015 1.293347 1.726209 3.156466
1.169022 -2.170857 2.501448 4.521917 -3.222670 -2.045799 2.140289

-1.084097 -0.158587 1.142086 0.619553 -0.387846 0.520420 0.739324
-0.436208 1.077760 -0.242432 0.613347 -0.508242 1.046495 -1.045720
0.715088 -0.134558 -0.689231 -0.823494 0.368909 -1.370038 1.315888

-1.612516 0.626688 0.497277 0.868476 -1.089145 0.466725 -0.176755
-0.405549 0.724633 0.316869 -0.395077 -0.572348 0.971917 0.215410
0.699812 -0.346523 0.190417 0.048514 -0.029621 -0.693561 0.642459

-1.519834 -2.091371 0.977768 1.102681 0.570099 0.128182 0.640579
0.942271 1.099445 0.924383 0.892113 0.870657 1.434432 -2.180854
1.096040 1.888541 -0.928790 -0.883472 1.542002 -0.416998 1.156461
```

```
-1.570367 -0.677793 0.226100 1.764017 -1.828677 1.567315 1.477878
1.148266 1.579716 1.199377 0.612508 -1.836390 1.500654 -0.249682
1.766058 0.133162 -0.958027 -0.374008 1.015859 -0.542014 1.751147

-1.907820 -0.394480 1.020382 2.535741 -0.701287 2.727886 2.527893
1.363658 3.060167 0.757021 0.808590 -1.068790 3.353714 -1.242597
2.873088 1.138384 -1.733479 -0.988560 1.121083 -2.334228 3.265811

-0.642424 -0.492139 -2.048815 -0.541581 -1.210931 0.951011 0.809358
0.919388 -0.486266 1.320528 0.485371 -0.676083 -0.066143 -0.793383
0.681467 0.587875 -0.704687 -1.470558 1.276990 1.443549 -0.598405

0.031436 0.110491 -1.843098 -0.723645 -0.732648 0.249876 0.005091
0.756025 -0.448720 0.654484 1.413580 -0.027333 -1.371305 -0.272806
-0.521199 0.513364 -0.566554 -1.670189 1.027185 1.767372 -0.954479

0.212456 -0.110232 -0.504643 -0.186376 -0.322148 0.533819 -0.009564
-0.294193 -0.319235 -0.041554 -0.316826 -0.340404 -0.138239 0.095486
0.223329 -0.060301 0.195037 0.502092 0.690255 -0.009003 -0.137465

-1.331791 -0.449366 0.499541 0.566375 -2.366744 0.717324 1.092386
0.513684 1.233986 0.752371 0.938478 -1.176524 0.456605 -0.643575
0.761009 0.611452 -0.383404 -0.571936 0.702037 1.154327 0.761880

-0.485158 -0.033434 -0.204256 -0.546265 -0.466986 0.500706 0.420496
0.175784 0.269771 0.234951 0.360819 -0.437277 -0.207511 0.139629
0.244054 0.503267 -0.074109 -0.546761 0.524598 0.163804 0.129772

1.698892 0.157734 0.207296 0.211463 0.855619 0.953850 0.372028
0.267671 0.217384 0.665358 0.883118 0.419388 0.156848 0.722571
-0.076201 0.912898 0.417693 1.040811 0.140338 0.358399 -0.348217

-1.204359 -0.173964 0.228355 0.016269 -0.620317 0.491706 0.778320
0.257104 0.474379 0.499612 0.505178 0.025653 0.235551 -0.200258
0.039131 0.188914 -0.226758 -0.530550 0.073820 0.308231 0.329993

-1.117419 -4.433354 -2.249738 -1.039407 0.079813 0.621007 1.126499
1.394437 -2.149711 1.109808 1.952027 -0.220723 -0.946343 -4.055124
0.322124 3.627469 -2.576343 -3.042104 1.683860 1.953913 -0.494628

-0.022051 0.360152 1.366324 1.035333 0.655870 -0.304117 -0.355162
-0.978796 -0.101089 -1.097993 -0.704499 -0.182840 0.345278 0.371454
-0.565152 -0.449148 0.529841 0.561098 -0.497486 -0.340474 -0.012436

-0.550508 -1.750732 1.572984 0.774377 0.059253 0.391510 0.174634
```

```
1.271411 0.578068 0.477621 1.543861 0.455551 1.126381 -1.572385
0.699392 0.958840 -1.203560 -1.389453 1.532846 0.195336 1.177601

-1.721479 0.066643 1.071155 1.324045 -0.257102 0.590952 0.750567
-0.494603 0.589836 -0.283417 -0.201645 -0.644957 1.120332 0.455839
0.792021 -0.711172 -0.016615 -0.015710 -0.113993 -0.927959 1.016164

-0.343042 0.366717 0.271766 0.702928 -0.817281 -0.043227 0.027978
0.551343 0.279888 0.395011 0.084472 -0.797986 0.088350 0.019563
-0.181188 -0.233073 -0.037661 0.545804 -0.432043 0.016452 0.031827

-6.507097 -0.135147 -0.863680 2.814351 -0.035294 0.244432 1.757570
1.667872 1.755216 -0.280799 -0.106896 0.536545 1.602686 0.288638
-3.852505 0.107994 -2.178458 -0.026141 1.465050 0.125630 0.221265

-1.121534 -0.517675 -0.535688 -2.507398 -0.928675 -0.109634 -1.528684
 2.328856 5.246791 -0.158578 0.584661 0.473670 0.479324 -0.014662
-0.100832 -0.149790 2.498991 0.154160 -1.370909 -0.347908 0.216538

-0.835554 -0.554393 -2.546383 0.623361 -0.651013 -1.573439 -0.943869
-1.507518 -2.429259 -1.513060 0.212532 0.255867 -1.318749 -0.389932
-0.680736 -0.810075 -3.360398 0.411270 -0.592674 -1.362175 -0.800838
```

Bibliography

Bibliography

[1] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with Splash 2. In *Proceedings 2nd IEEE Workshop on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 155–164, April, 1994.

[2] R. C. Agarwal, B. Alpern, L. Carter, F. G. Gustavson, D. J. Klepacki, R. Lawrence, and M. Zubair. High-performance parallel implementations of the NAS kernel benchmarks on the IBM SP2. *IBM Systems Journal*, 34(2):263–272, 1995.

[3] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir. SP2 system architecure. *IBM Systems Journal*, 34(2):152–184, 1995.

[4] Alacron, Nashua, New Hampshire. *Alacron FT 200 and Sharc*, 1995.

[5] R. Allen, D. Yasuda, S. Tanimoto, L. Shapiro, and L. Cinque. A parallel algorithm for graph matching and its MasPar implementation. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, New Orleans*, pages 13–18, December, 1993.

[6] H. M. Alnuweiri. Constant-time parallel algorithms for image labeling on a reconfigurable network of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):320–326, March 1994.

[7] H. M. Alnuweiri and V. K. Prasanna. Parallel architectures and algorithms for image component labeling. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(10):1014–1034, October 1992.

[8] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computer. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 32–38, April, 1995.

[9] N. Ansari, M.-H. Chen, and E. S. H. Hou. A genetic algorithm for point pattern matching. In B. Soucek, editor, *Dynamic, Genetic, and Chaotic Programming*, pages 353–371. John Wiley and Sons, New York, 1992.

[10] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–322, 1992.

[11] J. M. Arnold and M. A. McGarry. Splash 2 programmer's manual. Technical Report SRC-TR-93-107, Supercomputing Research Center, Bowie, Maryland, 1994.

[12] K. Asanovic, J. Beck, J. Feldman, N. Morgan, and J. Wawrzynek. A supercomputer for neural computation. In *Proc. Intl. Joint Conference on Neural Networks*, pages 5–9, Orlando, Florida, June 1994.

[13] K. Asanovic, J. Beck, B. E. D. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek. SPERT: A VLIW/SIMD neuro-microprocessor. In *Proc. Intl. Joint Conference on Neural Networks*, pages II–577–II–582, Baltimore, June 1992.

[14] P. M. Athanas and A. L. Abbott. Real-time image processing on a custom-computing platform. *IEEE Computer*, 28(2):16–24, February 1995.

[15] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[16] A. Averbuch, E. Gabber, B. Gordissky, and Y. Medan. A parallel FFT on an MIMD machine. *Parallel Computing*, 15(1-3):61–74, September 1990.

[17] H. S. Baird. *Model-Based Image Matching using Location.* The MIT Press, Cambridge, Massachusetts, 1985.

[18] D. H. Ballard and C. M. Brown. *Computer Vision.* Prentice-Hall, Englewood Cliffs, New Jersy, 1982.

[19] S. M. Barber, J. G. Delgado-Frias, S. Vassiliadis, and G. G. Pechanek. SPIN-L: sequential pipelined neuro-emulator with learning capabilities. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1927–1930, Nagoya, Japan, October 1993.

[20] A. Basu and C. M. Brown. Algorithms and hardware for efficient image smoothing. *Computer Vision, Graphics, and Image Processing*, 40(2):131–146, November 1987.

[21] D. Ben-Tzvi, A. Naqvi, and M. Sandler. Synchronous multiprocessor implementation of the Hough transform. *Computer Vision, Graphics, and Image Processing*, 52:437–446, 1990.

[22] P. Bertin and H. Touati. PAM programming environments: Practice and experience. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 133–139, April, 1994.

[23] S. Bhama, H. Singh, and N. D. Phadte. Parallelism for the faster implementation of the K-L transform for image compression. *Pattern Recognition Letters*, 14(8):651–659, August 1993.

[24] B. Bhanu and L. A. Nutall. Recognition of 3-D objects in range images using a Butterfly multiprocessor. *Pattern Recognition*, 22(1):49–64, Jan 1989.

[25] S. K. Bhaskar, A. Rosenfeld, and A. Y. Wu. Parallel processing of regions represented by quadtrees. *Computer Vision, Graphics, and Image Processing*, 42(3):371–381, June 1988.

[26] P. K. Biswas, J. Mukherjee, and B. N. Chaterji. Component labeling in pyramid architecture. *Pattern Recognition*, 26(7):1009–1115, July 1993.

[27] R. V. D. Boomgaard and R. V. Balen. Methods for fast morphological image transform using bitmapped binary images. *CVGIP: Graphical Models and Image Processing*, 54(3):252–258, May 1992.

[28] N. M. Bortos and M. Abdul-Aziz. Harware implementation of an artificial neural network. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1252–1257, Nagoya, Japan, October 1993.

[29] D. E. V. D. Bout, J. N. Morris, D. Thomae, S. Labrozzi, S. Wingo, and D. Hallman. Anyboard: An fpga-based reconfigurable system. *IEEE Design and Test of Computers*, pages 21–30, September 1992.

[30] C. M. Brown and D. Terzopoulos, editors. *Real-time computer vision.* Cambridge University Press, Cambridge, 1994.

[31] J. Brown and D. Crookes. A high level language for parallel image processing. *Image and Vision Computing*, 12(2):67–79, March 1994.

[32] D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, editors. *Splash 2: FPGAs for Custom Computing Machines.* IEEE Computer Society Press, Los Alamitos, 1996.

[33] C. Chakrabarti and J. Jaja. VLSI architectures for template matching and block matching. In V. K. P. Kumar, editor, *Parallel architectures and algorithms for image understanding*, pages 3–27. Academic Press, San Diego, 1991.

[34] P. K. Chan and S. Mourad. *Digital Design using Field Programmable Gate Arrays.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[35] J. H. Chang, O. H. Ibarra, T. Pong, and S. M. Sohn. Two-dimensional convolution on a pyramid computer. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(4):590–593, July 1988.

[36] V. Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[37] R. Chellappa and A. Rosenfeld. Vision engineering: Designing computer vision systems. In C. H. Chen, L. F. Pau, and P. S. P. Wang, editors, *Handbook of Pattern Recognition and Computer Vision*, pages 805–815. World Scientific Publishing Company, New Jersey, 1993.

[38] K. Chen. Efficient parallel algorithms for the computation of two-dimensional image moments. *Pattern Recognition*, 23(1/2):109–119, 1990.

[39] M.-H. Chen and T. Pavlidis. Image seaming for segmentation on parallel architecture. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(6):588–594, June 1990.

[40] Y.-A. Chen, Y.-L. Lin, and L.-W. Chang. A systolic algorithm for the k-nearest neighbors problem. *IEEE Trans. on Computers*, 41(1):103–108, January 1992.

[41] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20(1):125–141, 1987.

[42] H.-D. Cheng, W.-C. Lin, and K.-S. Fu. Space-time domain expansion approach to VLSI and its application to hierarchical scene matching. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-7(3):306–319, May 1985.

[43] H. D. Cheng, C. Tong, and Y. J. Lu. VLSI curve detector. *Pattern Recognition*, 23(1/2):35–50, 1990.

[44] G. Chinn, K. A. Grajski, C. Chen, C. Kuszmaul, and S. Tomboulian. Systolic array implementation of neural nets on the MasPar MP-1 massively parallel processor. In *Proc. Intl. Joint Conference on Neural Networks*, pages II–169–II–173, San Diego, 1990.

[45] A. Choudhary and S. Ranka. Mesh and pyramid algorithms for iconic indexing. *Pattern Recognition*, 25(9):1061–1067, May 1992.

[46] A. Choudhary and R. Thakur. Connected component labeling on coarse grain parallel computers: An experimental study. *Journal of Parallel and Distributed Computing*, 20(1):78–83, January 1994.

[47] A. N. Choudhary, J. H. Patel, and N. Ahuja. NETRA: A hierarchical and partitionable architecture for computer vision systems. *IEEE Trans. on Parallel and Distributed Systems*, 4(10):1092–1104, October 1993.

[48] K.-L. Chung and H.-Y. Lin. Hough transform on reconfigurable meshes. *CVGIP: Image Understanding*, 61(2):278–284, March 1995.

[49] Y. Chung, V. K. Prasanna, and C.-L. Wang. A fast asynchronous algorithm for linear feature extraction on IBM SP-2. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 294–301, September, 1995.

[50] L. Cinque, C. Guerra, and S. Levialdi. Computing shape description transform on a multiresolution architecture. *CVGIP: Image Understanding*, 55(3):287–295, May 1992.

[51] M. Conner and R. Tolimieri. Special purpose hardware for Discrete Fourier Transform implementation. *Parallel Computing*, 20(2):215–232, February 1994.

[52] C. E. Cox and E. Blanz. GANGLION–a fast field-programmable gate array implementation of a connectionist classifier. *IEEE Journal of Solid-State Circuits*, 27(3):288–299, March 1992.

[53] J. D. Crisman and J. A. Webb. The Warp machine on Navlab. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13(5):451–465, May 1991.

[54] R. Cypher, J. L. C. Sanz, and L. Snyder. An EREW PRAM algorithm for image component labeling. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(3):258–261, March 1989.

[55] M. Daallen, P. Jeavons, and J. Shawe-Taylor. A stochastic neural architecture that exploits dynamic reconfigurable FPGAs. In *Proceedings 1st IEEE Workshop on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 202–211, April, 1993.

[56] Data Translation, Marlboro, Massachusetts. *Data Translation Data book*, 1996.

[57] Datacube Inc., Massachusetts. *MaxVideo 250*, 1995.

[58] H. Derin and C.-S. Won. A parallel image segmentation algorithm using relaxation with varying neighborhoods and its mapping to array processors. *Computer Vision, Graphics, and Image Processing*, 40(1):54–78, October 1987.

[59] V. Dixit and D. I. Moldovan. Semantic network array processor and its applications to image understanding. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-9(1):153–160, January 1987.

[60] H. Embrechts, D. Roose, and P. Wambacq. Component labeling on a MIMD multiprocessor. *CVGIP: Image Understanding*, 57(2):155–165, March 1993.

[61] D. J. Evans and M. Gusev. New linear systolic arrays for digital filter and convolution. *Parallel Computing*, 20(1):29–61, January 1994.

[62] W. C. Fang, C. Y. Chang, B. J. Sheu, O. T. C. Chen, and J. C. Curlander. VLSI systolic binary tree-searched vector quantizer for image compression. *IEEE Transactions on Very Large Scale Integration Systems*, 2(1):33–44, March 1994.

[63] Z. Fang, X. Li, and L. M. Ni. Parallel algorithms for image template matching on hypercube SIMD computers. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9(6):835–841, November 1987.

[64] Z. Fang, X. Li, and L. M. Ni. On the communication complexity of generalized 2-d convolution on array processors. *IEEE Trans. on Computers*, 38(2):184–194, February 1989.

[65] Federal Bureau of Investigation, U. S. Government Printing Office, Washington, D. C. *The Science of Fingerprints: Classification and Uses*, 1984.

[66] O. Firschein. Defense applications of image understanding. *IEEE Expert*, pages 11–17, October 1995.

[67] A. L. Fisher and P. T. Highnam. Computing the Hough transform on a scan line array processor. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(3):262–265, March 1989.

[68] A. L. Fisher and P. T. Highnam. The SLAP image computer. In V. K. P. Kumar, editor, *Parallel Architectures and Algorithms for Image Understanding*, pages 307–337. Academic Press, San Diego, 1991.

[69] Y. Fujita, N. Yamashita, and S. Okazaki. A 64 parallel integrated memory array processor and a 30 GIPS real-time vision system. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 242–249, September, 1995.

[70] D. Galloway, D. Karchmer, P. Chow, D. Lewis, and J. Rose. The Transmogrifier: The University of Toronto Field-Programmable System. Technical report, CSRI-306, 1994.

[71] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's guide and reference manual.* Oak Ridge National Laboratory, Tennessee, 1993.

[72] L. Geppert. Technology 1995: Solid state. *IEEE Spectrum*, 32(1):35–39, January 1995.

[73] D. Gerogiannis and S. C. Orphanoudakis. Load balancing requirements in parallel implementations of image feature extraction tasks. *IEEE Trans. on Parallel and Distributed Systems*, 4(9):994–1013, September 1993.

[74] I. Gertner and M. Rofheart. A parallel algorithm for 2-D FFT computation with no interprocessor communication. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):377–382, July 1990.

[75] J. Ghosh and K. Hwang. Mapping neural networks onto message passing multicomputers. *Journal of Parallel and Distributed Computing*, 6:291–330, 1989.

[76] J. M. Gilbert and W. Yang. A real-time face recognition system using custom VLSI hardware. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, New Orleans*, pages 58–66, December, 1993.

[77] M. Gokhale and R. Minich. FPGA computing in a data parallel C. In *Proceedings 1st IEEE Workshop on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 94–102, 1993.

[78] M. Gokhale and B. Schott. Data parallel C on a reconfigurable logic array. Technical Report SRC-TR-94-121, Supercomputing Research Center, Bowie, Maryland, 1994.

[79] M. Gokmen and C.-C. Li. Edge detection and surface reconstruction using refined regularization. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 15(5):492–499, May 1993.

[80] R. C. Gonzalez and P. Wintz. *Digital Image Processing.* Addison-Wesley, Reading, Masschusetts, second edition, 1987.

[81] J. Greene, E. Hamdy, and S. Beal. Antifuse field programmable gate arrays. *Proceedings of the IEEE*, 81(7):1042–1056, July 1993.

[82] J. Gu, W. Wang, and T. C. Henderson. A parallel architecture for discrete relaxation algorithm. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-9(9):816–831, November 1987.

[83] A. Gupta and V. Kumar. The scalability of FFT on parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(8):922–932, August 1993.

[84] R. K. Gupta. Hardware-software cosynthesis for digital systems. *IEEE Design and Test of Computers*, pages 29–41, Spetember 1993.

[85] S. Hambrusch, X. He, and R. Miller. Parallel algorithms for gray-scale digitized picture component labelling on a mesh-connected computer. *Journal of Parallel and Distributed Computing*, 20(1):56–68, January 1994.

[86] M. Hamdi. Parallel architectures for wavelet transform. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, New Orleans*, pages 376–384, December, 1993.

[87] D. Hammerstrom. A VLSI architecture for high-performance, low-cost, on-chip learning. In *Proc. Intl. Joint Conference on Neural Networks*, pages II–537–II–544, San Diego, 1990.

[88] K. Hanahara, T. Maruyama, and T. Uchiyama. A real-time processor for the Hough transform. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(1):121–125, January 1988.

[89] R. M. Haralick. Document image undestanding: geometric and logical layout. In *Proc. of IEEE Computer Vision and Pattern Recognition, Seattle*, pages 385–390, June, 1994.

[90] R. M. Haralick and L. G. Shapiro. Image segmentation techniques. *Computer Vision, Graphics, and Image Processing*, 29(1):100–132, January 1985.

[91] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision.* Addison-Wesley, Reading, Masschusetts, 1993.

[92] R. M. Haralick, S. R. Sternberg, and X. Zhuang. Image analysis using mathematical morphology. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9(4):532–550, July 1987.

[93] D. Helman and J. Jaja. Efficient image processing algorithms on the scan line array processor. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17(1):47–56, January 1995.

[94] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, San Mateo, California, 1990.

[95] S. Heydorn and P. Weinder. Optimization and performance analysis of thinning algorithms on parallel computers. *Parallel Computing*, 17(1):17–27, April 1991.

[96] K. Hwang. *Advnaced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, 1993.

[97] H. A. H. Ibrahim, J. R. Kender, and D. E. Shaw. On the application of massively parallel simd tree machines to certain intermediate-level vision tasks. *Computer Vision, Graphics, and Image Processing*, 36(1):53–75, October 1986.

[98] IEEE Computer Society Press, Los Alamitos, California. *Proc. of FPGAs for custom computing machines, Napa Valley, California*, April, 1993.

[99] IEEE Computer Society Press, Los Alamitos, California. *Proc. of FPGAs for custom computing machines, Napa Valley, California*, April, 1994.

[100] IEEE Computer Society Press, Los Alamitos, California. *Proc. of FPGAs for custom computing machines, Napa Valley, California*, April, 1995.

[101] Imaging Technology, Bedford, Massachusetts. *MVC 160*, 1996.

[102] A. K. Jain. *Fundamentals of Digital Image Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[103] A. K. Jain and S. Bhattacharjee. Text segmentation using Gabor filters for automatic document processing. *Machine Vision and Applications*, 5:169–184, 1992.

[104] A. K. Jain and Y. Chen. Address block location using color and texture analysis. *CVGIP: Image Understanding*, 60(2):179–190, September 1994.

[105] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Englewood Cliffs, New Jersy, 1988.

[106] A. K. Jain and K. Karu. Learning texture discrimination masks. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 18(2):195–205, February 1996.

[107] A. K. Jain and Y. Zhong. Page layout segmentaion based on texture analysis. In *Proc. 2nd International Conf. on Image Processing, Washington, D. C.*, pages 308–311, October, 1995.

[108] J. Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Masschusetts, 1992.

[109] J. W. Jang, H. Park, and V. K. Prasanna. A fast algorithm for computing histogram on reconfigurable mesh. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17(2):97–106, February 1995.

[110] J. H. Jenkins. *Designing with FPGAs and CPLDs.* Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[111] J.-F. Jenq and S. Sahni. Image shrinking and expanding on a pyramid. *IEEE Trans. on Parallel and Distributed Systems*, 4(11):1291–1296, November 1993.

[112] J.-F. Jenq and S. Sahni. Reconfigurable mesh algorithms for the Hough transform. *Journal of Parallel and Distributed Computing*, 20(1):69–77, January 1994.

[113] S. L. Johnson and R. L. Krawitz. Cooley-Tukey FFT on the connection machine. *Parallel Computing*, 18(11):1201–1221, November 1992.

[114] J.-M. Jolion. Computer vision methodologies. *CVGIP: Image Understanding*, 59(1):53–71, January 1994.

[115] P. P. Jonker. Why linear arrays are better image processors? In *Proc. of 12th Int'l. Conf. on Pattern Recognition, Jerusalem*, pages 334–338, 1994.

[116] S. J. C. Jr., L.-P. Yuan, and R. Ehrlich. A fast and accurate erosion-dilation method suitable for microcomputers. *CVGIP: Graphical Models and Image Processing*, 53(3):283–290, May 1991.

[117] D. Judd, N. K. Ratha, P. K. McKinley, J. Weng, and A. K. Jain. Parallel implementation of vision algorithms on workstation clusters. In *Proc. of 12th Intl. Conf. of Pattern Recognition, Jerusalem*, pages 317–321, 1994.

[118] P. Kahn. Building blocks for computer vision systems. *IEEE Expert*, 8(6):40–50, December 1993.

[119] A. Kalavade and E. A. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design and Test of Computers*, pages 16–28, December 1993.

[120] M. Kamada, K. Toraichi, R. Mori, K. Yamamoto, and H. Yamada. A parallel architecture for relaxation operation. *Pattern Recognition*, 21(2):175–181, 1988.

[121] E. W. Kent, M. O. Shneier, and R. Lumia. PIPE. *Journal of Parallel and Distributed Computing*, 2:50–78, 1985.

[122] A. Khotanzad and A. Bourfa. Image segementation by a parallel, non-parametric histogram based clustering algorithm. *Pattern Recognition*, 23(9):961–973, September 1990.

[123] H. N. Kim, M. J. Irwin, and R. M. Owens. MGAP applications in machine perception. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 67–75, September, 1995.

[124] D. V. Kirsanov. Digital architecture for neural networks. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1939–1942, Nagoya, Japan, October 1993.

[125] P. Kotilainen, J. Saarinen, and K. Kaski. Neural network computation in a parallel multiprocessor architecture. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1979–1982, Nagoya, Japan, October 1993.

[126] O. G. Koufopavlou and C. E. Goutis. Image reconstruction on a special purpose array processor. *Image and Vision Computing*, 10(7):479–484, September 1992.

[127] A. V. Kulkarni and D. W. L. Yen. Systolic processing and implementation for signal and image processing. *IEEE Trans. on Computers*, 31(10):1000–1009, October 1982.

[128] V. K. P. Kumar and V. Krishnan. Efficient parallel algorithm for image template matching on hypercube SIMD machines. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(6):665–669, June 1989.

[129] H. T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, January 1982.

[130] H. T. Kung, L. M. Ruane, and D. W. L. Yen. A two-level pipelined systolic array for convolutions. In H. T. Kung, B. Sproull, and G. Steele, editors, *VLSI Systems and Computations*, pages 255–264. Computer Science Press, Maryland, 1981.

[131] D. Lattard and G. Mazare. A VLSI implementation of parallel image reconstruction. *CVGIP: Graphical Models and Image Processing*, 53(6):581–591, November 1991.

[132] H. C. Lee and R. E. Gaensslen, editors. *Advances in Fingerprint Technology.* Elsevier, New York, 1991.

[133] S.-W. Lee and W.-H. Hsu. Parallel algorithms for hidden markov models on the orthogonal multiprocessor. *Pattern Recognition*, 25(2):219–232, Feb 1992.

[134] S. Y. Lee and J. K. Aggarwal. Parallel 2-D convolution on a mesh connected array processor. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9(4):590–594, July 1987.

[135] S.-Y. Lee and J. K. Aggarwal. A system design/scheduling strategy for parallel image processing. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(2):194–204, February 1990.

[136] S. Y. Lee, S. Yalamanchili, and J. K. Aggarwal. Parallel image normalization on a mesh connected array processor. *Pattern Recognition*, 20(1):115–124, 1987.

[137] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, San Mateo, California, 1992.

[138] P. Lenders and H. Schroder. A programmable systolic device for image processing based on mathematical morphology. *Parallel Computing*, 13(3):337–344, March 1990.

[139] M. D. Levine. Nonmetric multidimensional scaling and hierarchical clustering – procedure for investigations of perception of sports. *Research Quarterly*, 48:341–348, 1977.

[140] H. F. Li, R. Jayakumar, and M. Youssef. Parallel algorithms for recognizing handwritten characters using shape features. *Pattern Recognition*, 22(6):641–652, 1989.

[141] X. Li and Z. Fang. Parallel clustering algorithms. *Parallel Computing*, 11(3):275–290, August 1989.

[142] S. Liang, M. Ahmadi, and M. Shridhar. A morphological approach to text string extraction from regular periodic overlapping text background images. *CVGIP: Graphical Models and Image Processing*, 56(5):402–413, September 1994.

[143] R. Lin and E. K. Wong. Logic gate implementation for gray-scale morphology. *Pattern Recognition Letters*, 13(7):481–487, July 1992.

[144] W.-M. Lin and V. K. P. Kumar. Efficient histogramming on hypercube SIMD machines. *Computer Vision, Graphics, and Image Processing*, 49:104–120, 1990.

[145] J. J. Little, G. E. Blelloch, and T. A. Cass. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(3):244–257, March 1989.

[146] X. Liu and G. L. Wilcox. Benchmarking of the CM-5 and Cray machines with a very large backpropagation neural network. In *Proc. Intl. Joint Conference on Neural Networks*, pages 22–27, Orlando, Florida, June 1994.

[147] D. G. Lowe. *Perceptual Organization and Visual Recognition.* Kluwer Academic Publishers, Massachussets, 1985.

[148] M. Manohar and H. K. Ramapriyan. Connected component labeling of binary images on a mesh connected massively parallel processor. *Computer Vision, Graphics, and Image Processing*, 45:133–149, 1989.

[149] P. Maragos. Tutorial on advances in morphological image processing and analysis. *Optical Engineering*, 26(7):623–632, July 1987.

[150] M. Maresca, M. A. Lavin, and H. Li. Parallel architectures for vision. *Proceedings of the IEEE*, 76(8):970–981, August 1988.

[151] D. Marr. *Vision*. W. H. Freeman and Co., San Francsisco, 1982.

[152] P. Mass, K. Hoen, and H. Wallinga. 70 input, 20 nanosecond pattern classifier. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1854–1859, Orlando, Florida, June 1994.

[153] R. W. Means. High speed parallel hardware performance issues for neural network applications. In *Proc. Intl. Joint Conference on Neural Networks*, pages 10–16, Orlando, Florida, June 1994.

[154] R. S. Michaleski and R. E. Stepp. Automated construction of classifications: Conceptual clustering versus numerical taxonomy. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 5(4):396–409, July 1983.

[155] G. D. Micheli. Computer-aided Hardware-Software codesign. *IEEE Micro*, pages 10–16, August 1994.

[156] B. Miller. Vital signs of identity. *IEEE Spectrum*, 31(2):22–30, February 1994.

[157] D. Mueller and D. Hammerstorm. A neural network system component. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1258–1264, Baltimore, June 1992.

[158] U. A. Muller. A high performance neural net simulation. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1–4, Orlando, Florida, June 1994.

[159] P. J. Narayanan and L. S. Davis. Replicated data algorithms in image processing. *CVGIP: Image Understanding*, 56(3):351–365, November 1992.

[160] L. M. Ni and A. K. Jain. A VLSI systolic architecture for pattern clustering. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-7(1):80–89, January 1985.

[161] T. Nordstrom and B. Svensson. Using and designing massively parallel computers for artificial neural networks. *Journal of Parallel and Distributed Computing*, 14:260–285, 1992.

[162] NSF. *Grand Challenge: High-Performance Computing and Communications.* Report of the Committee on Physical, Mathematical, and Engineering sciences, U. S. Office of Science and Technology Policy, National Science Foundation, Washington, D. C., 1992.

[163] S. Olariu, J. L. Schwing, and J. Zhang. Fast computer vision algorithms for reconfigurable meshes. *Image and Vision Computing*, 10(9):610–616, November 1992.

[164] S. Olariu, J. L. Schwing, and J. Zhang. Computing the Hough transform on reconfigurable meshes. *Image and Vision Computing*, 11(10):623–628, December 1993.

[165] S. Olariu, J. L. Schwing, and J. Zhang. Fast component labeling and convex hull computation on reconfigurable meshes. *Image and Vision Computing*, 11(7):447–455, September 1993.

[166] J. Onuki, T. Maenosono, M. Shibata, N. Iijima, H. Mitsui, and Y. Yoshida. ANN accelarotor by parallel processor based on DSP. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1913–1916, Nagoya, Japan, October 1993.

[167] S. Oteki, A. Hashimoto, T. Furuta, S. Motomura, T. Wantanabe, D. G. Stork, and H. Eguichi. A digital neural network VLSI with on-chip learning using stochastic pulse encoding. In *Proc. Intl. Joint Conference on Neural Networks*, pages 3039–3045, Nagoya, Japan, October 1993.

[168] N. R. Pal and S. K. Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277–1294, September 1993.

[169] J. N. Patel, A. A. Khokhar, and L. H. Jameison. Implementation of parallel image processing algorithms in the CLONER environment. In *Proc. of IEEE Workshop on VLSI signal processing, La Jolla, California*, pages 83–92, Oct., 1994.

[170] T. Pavlidis and J. Zhou. Page segmentation and classification. *CVGIP: Image Understanding*, 54(6):484–486, November 1992.

[171] G. G. Pechanek, S. Vassiliadis, J. G. Delgado-Frias, and G. Triantafyllos. Scalable completely connected digital neural network. In *Proc. Intl. Joint Conference on Neural Networks*, pages 2078–2083, Orlando, Florida, June 1994.

[172] J. B. Peterson and P. M. Athanas. Addressing the computational needs of high-speed image processing with a custom computing machine. *Journal of VLSI Signal Processing*. Under review.

[173] T. D. S. Pierre and M. Milgram. New and efficient cellular algorithms for image processing. *CVGIP: Image Understanding*, 55(3):261–274, May 1992.

[174] T. Poggio. Early vision: From computational structure to algorithms and parallel hardware. *Computer Vision, Graphics, and Image Processing*, 31(2):139–155, August 1985.

[175] V. K. Prasanna, C.-L. Wang, and A. A. Khokhar. Low level vision processing on connection machine CM-5. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, New Orleans*, pages 117–126, December, 1993.

[176] U. Ramacher. SYNAPSE—a neurocomputer that synthesizes neural algorithms on a parallel systolic engine. *Journal of Parallel and Distributed Computing*, 14:306–318, 1992.

[177] N. Ranganathan and K. B. Doreswamy. A VLSI chip for computing medial axis transform of an image. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 36–43, September, 1995.

[178] N. Ranganathan and R. Mehrotra. A VLSI architecture for dynamic scene analysis. *CVGIP: Image Understanding*, 53(2):189–197, March 1991.

[179] N. Ranganathan and M. Shah. A VLSI architecture for computing scale space. *Computer Vision, Graphics, and Image Processing*, 43:178–204, 1988.

[180] N. Ranganathan and S. Venugopal. An efficient VLSI architecture for template matching based on momemt preserving pattern matching. In *Proc. of 12th Int'l. Conf. on Pattern Recognition, Jerusalem*, pages 388–390, 1994.

[181] S. Ranka and S. Sahni. Convolution on mesh connected multicomputers. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12(3):315–318, March 1990.

[182] A. R. Rao. *A Taxonomy for Texture Description and Identification.* Springer-Verlag, New York, 1990.

[183] N. K. Ratha, T. Acar, M. Gokmen, and A. K. Jain. A distrbuted edge detection and surface reconstruction algorithm based on weak membrane modeling. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 149–154, September, 1995.

[184] N. K. Ratha, S. Chen, and A. K. Jain. Adaptive flow orientation based texture extraction in fingerprint images. *Pattern Recognition*, 28(11):1657–1672, November 1995.

[185] N. K. Ratha and A. K. Jain. High performance custom computing for image segmentation. In *High Performance Computing Conference, New Delhi*, pages 67–72, December, 1995.

[186] N. K. Ratha, A. K. Jain, and M. J. Chung. Clustering using coarse-grained parallel genetic algorithm: a preliminary study. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 331–338, September, 1995.

[187] N. K. Ratha, A. K. Jain, and D. T. Rover. Convolution on Splash 2. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California*, pages 204–213, 1995.

[188] N. K. Ratha, A. K. Jain, and D. T. Rover. Fpga-based high performance page layout segmentation. In *Proc. of the IEEE Great Lakes Symposium on VLSI, Ames, Iowa*, pages 29–34, March, 1996.

[189] N. K. Ratha, A. K. Jain, and D. T. Rover. An FPGA-based point pattern matching coprocessor with application to fingerprint matching. In *Proc. of IEEE Workshop on Computer Architecture for Machine Perception, Como, Italy*, pages 394–401, September, 1995.

[190] J. H. Reif, editor. *Synthesis of Parallel Algorithms.* Morgan Kaufman, San Mateo, California, 1993.

[191] F. M. Rhodes, J. J. Dituri, G. H. Chapman, B. E. Emerson, A. M. Soares, and J. I. Raffel. A monolithic Hough transform processor based on restructurable VLSI. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 10(1):106–110, January 1988.

[192] I. Rigoutsos and R. Hummel. Massively parallel model matching. *IEEE Computer*, 25(2):33–42, February 1992.

[193] J. Rose, A. E. Gammal, and A. Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, July 1993.

[194] A. Rosenfeld. Computer vision: Basic principles. *Proceedings of the IEEE*, 76(8):863–868, August 1988.

[195] E. Sackinger and H.-P. Graf. A board system for high-speed image analysis and neural networks. *IEEE Trans. on Neural Networks*, 7(1):214–221, January 1996.

[196] T. Sakai, M. Nagao, and H. Matsushima. Extraction of invariant picture sub-structures by computer. *Computer Graphics and Image Processing*, 1(1):81–96, 1972.

[197] A. Sangiovanni-Vincentelli, A. E. Gamal, and J. Rose. Synthesis methods for field programmable gate arrays. *Proceedings of the IEEE*, 81(7):1057–1083, July 1993.

[198] Y. Sato, K. Shibata, M. Asai, M. Ohki, M. sugie, T. Sakaguchi, M. Hashimoto, and Y. Kuwabara. Development of a high-performance general purpose neuro-computer composed of 512 digital neurons. In *Proc. Intl. Joint Conference on Neural Networks*, pages 1967–1970, Nagoya, Japan, October 1993.

[199] R. J. Schalkoff. *Digital Image Processing and Computer Vision.* John Wiley, New York, 1989.

[200] N. B. Serbedzija. Simulating artificial neural networks on parallel architecture. *IEEE Computer*, 29(3):56–63, March 1996.

[201] R. V. Shankar and S. Ranka. Parallel vision algorithms using sparse array presentations. *Pattern Recognition*, 26(10):1511–1519, October 1993.

[202] SharpImage Software, New York. *The HIPS Image Processing Software*, 1993.

[203] Y. Shimokawa, Y. Fuwa, and N. Aramaki. A parallel ASIC VLSI neural computer for a large number of neurons and billion connections per second speed. In *Proc. Intl. Joint Conference on Neural Networks*, pages 2162–2167, Seattle, July 1991.

[204] K. G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 82(1):6–24, January 1994.

[205] D. Skea, I. Barrodale, R. Kuwahara, and R. Poecker. A control point matching algorithm. *Pattern Recognition*, 26(2):269–276, Feb 1993.

[206] A. Srivastava. A comparison between conceptual clustering and conventional clustering. *Pattern Recognition*, 23(9):975–981, September 1990.

[207] X.-H. Sun and J. L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, 1991.

[208] M. H. Sunwoo and J. K. Aggarwal. Flexibly coupled multiprocessors for image processing. *Journal of Parallel and Distributed Computing*, 10(2):115–129, 1990.

[209] M. H. Sunwoo and J. K. Aggarwal. VisTA – An image understanding architecture. In V. K. P. Kumar, editor, *Parallel Architectures and Algorithms for Image Understanding*, pages 121–153. Academic Press, San Diego, 1991.

[210] M. F. X. B. V. Swaaij, F. V. M. Catthoor, and H. J. DeMan. Deriving ASIC architectures for the Hough transform. *Parallel Computing*, 16(1):113–121, November 1990.

[211] P. N. Swartztrauber, R. A. Sweet, W. L. Briggs, V. E. Henson, and J. Otto. Bluestein's FFT for arbitrary n on the hypercube. *Parallel Computing*, 17(6-7):607–617, September 1991.

[212] H. L. Tan, S. B. Gelfand, and E. J. Delp. A cost minimization approach to edge detection using simulated annealing. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(1):3–18, January 1991.

[213] S. L. Tanimoto and E. W. Kent. Architectures and algorithms for iconic-to-symbol transformation. *Pattern Recognition*, 23(12):1377–1388, December 1990.

[214] Texas Instruments, Texas. *Designer's workbook – MVP-80 training manual*, 1995.

[215] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, pages 6–15, September 1993.

[216] M. S. Tomlinson, D. J. Walker, and M. A. Sivilotti. A digital neural network architecture for VLSI. In *Proc. Intl. Joint Conference on Neural Networks*, pages II–545–II–550, San Diego, 1990.

[217] J. Ton and A. K. Jain. Registering Landsat images by point matching. *IEEE Trans. on Geoscience and Remote Sensing*, 27(5):642–651, September 1989.

[218] A. Torrabla. A systolic array with applications to image processing and wire-routing in VLSI circuits. *Parallel Computing*, 17(1):85–93, April 1991.

[219] N. Tredennick. Technology and business: forces driving microprocessor evolution. *Proceedings of the IEEE*, 83(12):1641–1652, December 1995.

[220] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, July 1993.

[221] S. Umeyama. Parameterized point pattern matching and its application to recognition of object families. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 15(2):136–144, February 1993.

[222] V. V. Vinod and S. Ghose. Point matching using asymmetrical neural networks. *Pattern Recognition*, 8(26):1207–1214, August 1993.

[223] M. A. Viredaz and P. Ienne. MANTRA I: a systolic neuro-computer. In *Proc. Intl. Joint Conference on Neural Networks*, pages 3054–3061, Nagoya, Japan, October 1993.

[224] C.-L. Wang, V. K. Prasanna, H. J. Kim, and A. A. Khokhar. Scalable data parallel implementations of object recognition using geometric hashing. *Journal of Parallel and Distributed Computing*, 21(1):96–109, April 1994.

[225] C. C. Weems. Architectural requirements of image understanding with respect to parallel processing. *Proceedings of the IEEE*, 79(4):537–547, April 1991.

[226] C. C. Weems, S. P. Levitan, A. R. Hanson, E. M. Riseman, D. B. Shu, and J. G. Nash. The image understanding architecture. *International Journal of Computer Vision*, 2(3):251–282, 1989.

[227] J. H. Wegstein. An automated fingerprint identification system. Technical Report 500-89, National Bureau of Standards, Bethesda, Maryland, 1982.

[228] F. Weil, L. H. Jamieson, and E. J. Delp. Dynamic intelligent scheduling and control of reconfigurable parallel architectures for computer vision/image processing. *Journal of Parallel and Distributed Computing*, 13(3):273–285, November 1991.

[229] J. Weng, N. Ahuja, and T. S. Huang. Matching two perspective views. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(8):806–825, August 1992.

[230] W. H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July 1994.

[231] J. Worlton. Toward a taxonomy of performance metrics. *Parallel Computing*, 17:1073–1092, 1991.

[232] A. Y. Wu, S. K. Bhaskar, and A. Rosenfeld. Parallel processing of region boundaries. *Pattern Recognition*, 22(2):165–172, 1989.

[233] A. Y. Wu and A. Rosenfeld. Parallel processing of encoded bit strings. *Pattern Recognition*, 21(6):559–565, 1988.

[234] Xilinx, Inc., San Jose, California. *The Programmable Logic Data Book*, 1994.

[235] S. Yalamachili and J. K. Aggarwal. A system organization for parallel image processing. *Pattern Recognition*, 18(1):17–29, 1985.

[236] D.-L. Yang and C.-H. Chen. A real-time systolic array for distance transform. In *Proc. of 12th Int'l. Conf. on Pattern Recognition, Jerusalem*, pages 342–344, 1994.

[237] J. C. Yen, F. J. Chang, and S. Chang. A new architecture for motion-compensated image coding. *Pattern Recognition*, 25(4):357–366, April 1992.

[238] E. L. Zapata, F. F. Rivera, and O. G. Plata. Parallel fuzzy clustering on fixed size hypercube SIMD computers. *Parallel Computing*, 11(3):291–303, August 1989.