CHAPTER

26 Real-Time Hardware and Systems Design Considerations

In general, vision involves huge amounts of computation, as images are two-dimensional and in real-time applications are liable to arrive at rates of 10–20 per second. Although humans are able to cope easily with these data rates, they are often beyond the processing capabilities of conventional computers. This chapter explores the situation, and demonstrates how, using special computational hardware, the processing problems can be tackled and alleviated.

Look out for:

- how parallel processing can radically improve the speed at which vision algorithms run.
- the concept of a SIMD (single instruction stream, multiple data stream) computer, with one processor per pixel in a 2-D array.
- Flynn's classification of sequential and parallel computers.
- how a vision algorithm may optimally be partitioned between hardware and software.
- modern real-time hardware options.
- the increasingly important status of FPGAs and GPUs in real-time hardware design.
- vision system design considerations and the optimization process.

Much of this book has been devoted to the systematic design of vision algorithms, and of necessity has tended to focus on a great variety of sub-problems such as edge detection. However, when embarking upon design of a *complete* vision system, the situation is much less "clean," and indeed is subject to all sorts of financial and marketing constraints, as well as fiercely nonideal data. In this context, vision system design is as much an art as a science, and is more subject to cyclic improvement than in an ideal world—as the last few sections of this chapter aim to indicate. Suffice it to say here that the situation must be viewed realistically with an eye to improvement.

26.1 INTRODUCTION

In Chapter 1, we started by pointing out that of the five senses, vision has the advantage of providing enormous amounts of information at very fast rates: this was observed to be useful to humans and should also be of great value with robots and other machines. However, the input of large quantities of data necessarily implies large amounts of data processing, and this can create problems—especially in real-time applications. Hence, it is no wonder that speed of processing has been alluded to on numerous occasions in earlier chapters of this book.

It is now necessary to examine how serious this situation is and to suggest what can be done to alleviate the problem. Consider a simple situation where it is necessary to examine products of size 64×64 pixels moving at rates of 10–20 per second along a conveyor: this amounts to a requirement to process up to 100,000 pixels per second—or typically four times this rate if space between objects is taken into account. In fact, the situation can be significantly worse than indicated by these figures. First, even a basic process such as edge detection generally requires a neighborhood of at least 9 pixels to be examined before an output pixel value can be computed: thus, the number of pixel memory accesses is already 10 times that given by the basic pixel processing rate. Second, functions such as skeletonization or size filtering require a number of basic processes to be applied in turn: e.g., eliminating objects up to 20 pixels wide requires 10 erosion operations, whereas thinning similar objects using simple "north-south-east-west" algorithms requires at least 40 whole-image operations. Third, typical applications such as inspection require a number of tasks—edge detection, object location, surface scrutiny, and so on—to be carried out. All these factors mean that the overall processing task may involve anything between 1 and 100 million pixels or other memory accesses per second. Finally, this analysis ignores the complex computations required for some types of 3-D modeling, and for certain more abstract processing operations (see Chapters 14 and 15), although the expanding area of hyperspectral imaging (Chapter 25) also has extremely demanding computational requirements.

These formidable processing requirements imply a need for very carefully thought out algorithm strategies. This means that special hardware will normally be needed[1] for almost all real-time applications (exceptions might occur in those tasks where performance rates are governed by the speed of a robot, vehicle, or other slow mechanical device). Broadly speaking, there are two main strategies for improving processing rates. The first involves employing a single very fast

[1]Sometime in the future, this view will need to be reconsidered: see Section 26.7.

processor that is fabricated using advanced technology, e.g., with gallium arsenide semiconductor devices, Josephson junction devices or perhaps optical processing elements (PEs). Such techniques can be expected to yield speed increases by a factor of 10 or so, which might be sufficiently rapid for certain applications. However, it is more likely that a second strategy will have to be invoked—that of parallel processing: this involves employing N processors working in parallel, thereby giving the possibility of enhancing speed by a factor N. This second strategy is particularly attractive: to achieve a given processing speed, it should be necessary only to increase the number of processors appropriately—although it has to be accepted that cost will be increased by a factor of around N, as for the speed. Clearly, it is partly a matter of economics which of the two strategies will be the better choice but at any time the first strategy will be technology-limited, whereas parallel processing seems more flexible and capable of giving the required speed in any circumstances. Hence, for the most part, parallel processing is considered in what follows.

26.2 PARALLEL PROCESSING

There are two main approaches to parallel processing: in the first, the computational *task* is split into a number of functions, which are then implemented by different processors and in the second, the *data* are split into several parts and different processors handle the different parts. These two approaches are sometimes called *algorithmic parallelism* and *data parallelism*, respectively. Note that if the data are split, different parts of the data are likely to be nominally similar, so there is no reason to make the PEs different from each other. However, if the task is split functionally, the functions are liable to be very different and it is most unlikely that the PEs should be identical.

The example cited in Section 26.1 involves a fixed sequence of processes being applied in turn to the input images. On the whole, this type of task is well adapted to algorithmic parallelism and indeed to being implemented as a pipelined processing system, each stage in the pipeline performing one task such as edge detection or thinning (Fig. 26.1) (note that each stage of a thinning task will probably have to be implemented as a single stage in the pipeline). Clearly, such an approach lacks generality but it is cost-effective in a large number of applications, since it is capable of providing a speedup factor of around two orders of magnitude without undue complexity. Unfortunately, this approach is liable to be inefficient in practice. This is because the speed at which the pipeline operates is dictated by the speed of the slowest device on the pipeline—faster speeds of the other stages constitute wasted computational capability. Variations in the data passing along the pipeline add to this problem: e.g., a wide object would require many passes of a thinning operation, so either thinning would not proceed to completion (and the effect of this would have to be anticipated and allowed for), or

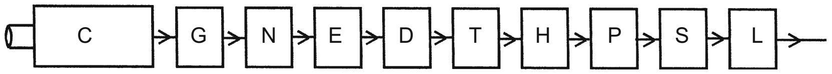

FIGURE 26.1

Typical pipelined processing system: C, input from camera; G, grab image; N, remove noise; E, enhance edge; D, detect edge; T, thin edge; H, generate Hough transform; P, detect peaks in parameter space; S, scrutinize products; L, log data and identify products to be rejected.

else the pipeline would have to be run at a slower rate. Obviously, it is necessary for such a system to be designed in accordance with worst-case rather than "average" conditions—although additional buffering between stages can help to reduce this latter problem.

Clearly, the design and control of a reliable pipelined processor is not trivial but, as mentioned above, it gives a generally cost-effective solution in many types of application. However, both with pipelined processors and with other machines that use algorithmic parallelism, there are significant difficulties in dividing tasks into functional partitions that match well the PEs on which they are to run. For this and other reasons, there have been many attempts at the alternative approach of data parallelism. Indeed, image data are, on the whole, reasonably homogeneous, so it is evidently worth searching for solutions incorporating data parallelism. Further consideration then leads to the SIMD type of machine in which each pixel is processed by its own PE. This method is described in Section 26.3.

26.3 SIMD SYSTEMS

In the SIMD (single instruction stream, multiple data stream) architecture, a 2-D array of PEs is constructed, which maps directly onto the image being processed; thus, each PE stores its own pixel value, processes it, and stores the processed pixel value. Furthermore, all PEs run the same program and indeed are subject to the same clock; this means that they execute the same instruction simultaneously (hence the existence of a single instruction stream). An additional feature of SIMD machines that are used for image processing is that each PE is connected to its immediate neighbors in the array, so that neighborhood operations can conveniently be carried out—the required input data are always available. This means that each PE is typically connected to eight others in a square array. Such machines therefore have the advantage not only of *image parallelism* but also of *neighborhood parallelism*—data from neighboring pixels are available immediately and several sequential memory accesses per pixel process are no longer required (for a useful review of these and other types of parallelism, see Danielsson and Levialdi (1981)).

The SIMD architecture is extremely attractive in principle since its processing structure seems closely matched to the requirements of many tasks, such as noise removal, edge detection, thinning, size analysis, and so on (although we return to this point below). However, in practice, it suffers from a number of disadvantages. Some of these are due to the compromises needed to keep costs at reasonable levels. For example, the PEs may not be powerful floating-point processors and may not contain much memory (this is because available cost is expended on including more PEs rather than making them more powerful); in addition, the processor array may be too small to handle the whole image at once, and problems of continuity and overlap arise when trying to process subimages separately (Davies et al., 1995); this can also lead to difficulties when global operations (such as finding an accurate convex hull) have to be performed on the whole image. Finally, getting the data in and out of the array can be a relatively slow process.

Although SIMD machines may appear to operate efficiently on image data, this is not always the case in practice, since many processors may be "ticking over" and not doing anything useful. For example, if a thinning algorithm is being implemented, much of the image may be bare of detail for most of the time, since most of the objects will have shrunk to a fraction of their original area. Thus, the PEs are not being kept *usefully* busy. Here the topology of the processing scheme is such that these inactive PEs are unable to get data they can act on, and efficiency drops off markedly. Hence, it is not obvious that an SIMD machine can always carry out the *overall* task any faster than a more modest MIMD machine (see definition and full explanation in Section 26.5), or a specially fast but significantly cheaper single processor (SISD) machine.

A more important characteristic is that although the SIMD machine is reasonably well adapted for image processing, it is quite restricted in its capabilities for image analysis. For example, it is virtually impossible to use *efficiently* for implementing Hough transforms, especially when these demand mapping image features into an abstract parameter space. In addition, most serial (SISD) computers are much more efficient at operations such as simple edge tracking, since their single processors are generally much faster than costs will permit for the many processors in an SIMD machine. Overall, these problems should be expected since the SIMD concept is designed for image-to-image transformations *via* local operators and does not map well to (a) image-to-image transformations that demand *nonlocal* operations, (b) image-to-abstract data transforms (intermediate-level processing), or (c) abstract-to-abstract data (high-level) processing (note that some would classify (a) as being a form of intermediate-level processing where *deductions* are made about what is happening in distant parts of an image—i.e., higher level interpretive data are being marked in the transformed image). This means that unaided SIMD machines are unlikely to be well suited for practical applications such as inspection.

Before leaving the topic of SIMD machines, recall that they incorporate two types of parallelism—image parallelism and neighborhood parallelism. Both of these contribute to high processing rates. Although it might at first appear that image parallelism contributes mainly through the high processing bandwidth[2] it offers, it also contributes through the high data accessing bandwidth: in contrast, neighborhood parallelism contributes only through the latter mechanism. However, what is important is that this type of parallel machine, in common with any successful parallel machine, incorporates both features. It is of little use to attend to the problem of achieving high processing bandwidth only to run into data bottlenecks through insufficient attention to data structures and data access rates: i.e., it is necessary to match the data access and processing bandwidths if full use is to be made of available processor parallelism.

26.4 THE GAIN IN SPEED ATTAINABLE WITH *N* PROCESSORS

It is interesting to speculate whether the gain in processing rate could ever be greater than N, say N^2. It could in principle be imagined that two robots used to make a bed would operate more efficiently than one, or four more efficiently than two, for a rectangular bed. Similarly, N robots welding N sections of a car body would operate more efficiently than a single one. The same idea should apply to N processors operating in parallel on an N-pixel image. At first sight, it does appear that a gain greater than N could result. However, closer study shows that any task is split between data organization and actual processing. Thus, the maximum gain that could result from the use of N processors is (exactly) N: any other factor is due to the difficulty, either for low or for high N, of getting the right data to the right processor at the right time. Thus, in the case of the bed-making robots, there is an overhead for $N = 1$ of having to run around the bed at various stages because the data (the sheets) are not presented correctly. More usually, it is at large N that the data are not available at the right place at the right moment. An immediate practical example of these ideas is that of accessing all eight neighbors in a 3×3 neighborhood where only four are directly connected, and the corner pixels have to be accessed *via* these four: then a *threefold* speedup in data access may be obtained by *doubling* the number of local links from four to eight.

There have been many attempts to model the utilization factor of both SIMD and pipelined machines when operating on branching and other algorithms. Minsky's conjecture (Minsky and Papert, 1971), that the gain in speed from a parallel processor is proportional to $\log_2 N$ rather than N, can be justified on this basis, and leads to an efficiency $\eta = \log_2 N/N$. Hwang and Briggs (1984) produced a more optimistic estimate of efficiency in parallel systems: $\eta = 1/\log_2 N$.

Following Chen (1971), the efficiency of a pipelined processor is usually estimated as $\eta = P/(N + P - 1)$, where there are on average P consecutive data

[2]In this context, it is conventional to use the term *bandwidth* to mean the maximum rate realizable *via* the stated mechanism.

points passing through a pipeline of N stages—the reasoning being based on the proportion of stages that are usefully busy at any one time. For imaging applications, such arguments are often somewhat irrelevant since the total delay through the pipelined processor is unimportant compared with the cycle time between successive input or output data values. This is because a machine that does not keep up with the input data stream will be unacceptable, whereas one that incorporates a fixed time delay may be acceptable in some cases (such as a conveyor belt inspection problem) although inadequate in others (such as a missile guidance system).

Broadly speaking, the situation that is being described here involves a speedup factor N coupled with an efficiency η, giving an overall speedup factor of $N' = \eta N$. Ultimately, the loss in efficiency is often due to frustrated algorithm branching processes but presents itself as underutilization of resources, which cannot be reduced because the incoming data are of variable complexity.

26.5 FLYNN'S CLASSIFICATION

Early in the development of parallel processing architectures, Flynn (1972) developed a now well-known classification, which has already been referred to above: architectures are either SISD, SIMD, or MIMD. Here SI (single instruction stream) means that a single program is employed for all the PEs in the system, whereas MI (multiple instruction stream) means that different programs can be used; SD (single data stream) means that a single stream of data is sent to all the PEs in the system, whereas MD (multiple data stream) means that the PEs are fed with data independently of each other.

The SISD machine is a single processor and is normally taken to refer to a conventional von Neumann computer. However, the definitions given above imply that SISD falls more naturally under the heading of a Harvard architecture, whose instructions and data are fed to it through separate channels: this gives it a degree of parallelism and makes it generally faster than a von Neumann architecture (in fact, there is almost invariably bit *parallelism* also, the data taking the form of words of data holding several bits of information, and the instructions being able to act on all bits simultaneously: however, this possibility is so universal that it will be accepted as standard in what follows).

The SIMD architecture has already been described reasonably thoroughly, although it is worth reiterating that the multiple data stream arises in imaging work through the separate pixels being processed by their PEs independently as separate, although similar, data streams. Note that the PEs of SIMD machines invariably embody the Harvard architecture.

The MISD architecture is notably absent from the above classification, although it is possible to envisage that pipelined processors fall into this category since a single stream of data is fed through all processors in turn, albeit being modified as it proceeds so the same *data* (as distinct from the same data stream) do not pass through each PE. However, many parties take the MISD category to be null (e.g., Hockney and Jesshope, 1981).

The MIMD category is a very wide one, containing all possible arrangements of separate PEs that get their data and their instructions independently: it even includes the case where none of the PEs are connected together in any way. However, such a wide interpretation does not solve practical problems. We can therefore envisage linking the PEs together by a common memory bus, or every PE being connected to every other one, or linkage by some other means. A common memory bus would tend to cause severe contention problems in a fast-operating parallel system, whereas maintaining separate links between all pairs of processors is clearly at the opposite extreme but would run into a combinatorial explosion as systems become larger. Hence, a variety of other arrangements is used in practice. Crossbar, star, ring, tree, pyramid, and hypercube structures have all been used (Fig. 26.2). In the crossbar arrangement, half of the processors can communicate directly with the other half *via* N links (and $N^2/4$ switches), although all processors can communicate with each other *indirectly*. In the star, there is one central PE so the maximum communication path has length 2. In the ring, all N PEs are placed symmetrically and the maximum communication path is of length $N - 1$ (note that this figure assumes unidirectional rings, which are easier to implement, and a number of notable examples of this type exist). In the

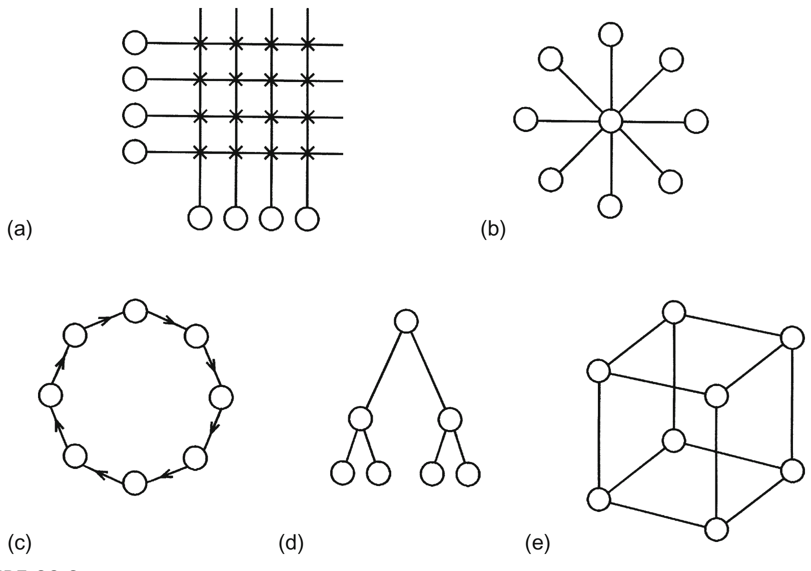

FIGURE 26.2

Possible arrangements for linking processing elements: (a) crossbar, (b) star, (c) ring, (d) tree and (e) hypercube. All links are bidirectional except where arrows indicate otherwise.

tree or pyramid, the maximum path length is of order $2(\log_2 N - 1)$, assuming that there are two branches for each node of the tree. Finally, for the hypercube of n dimensions (built in an n-dimensional space with two positions per dimension), the shortest path length between any two PEs is at most n; in this case, there are 2^n processors, so the shortest path length is at most $\log_2 N$. Overall, the basic principle must be to minimize path length while cutting down the possibility of data bottlenecks: this shows that the star configuration is very limited and explains the considerable attention that has been paid to the hypercube topology. In view of what was said earlier about the importance of matching the data bandwidth to the processing bandwidth, a very careful choice clearly needs to be made concerning a suitable architecture—and there is no lack of possible candidates (the above set of examples is by no means exhaustive).

Finally, note that factors other than speed enter into the choice of an architecture. For example, many have argued that the pyramid type of architecture closely matches the hierarchical data structures most appropriate for scene interpretation.

Much of the above discussion about architectures has been in the realm of the possible and the ideal (many of the existing systems are expensive experimental ones), and it is now necessary to consider more practical issues. How, e.g., are we to match the architecture to the data? More particularly, how do we lay down general guidelines for partitioning the tasks and implementing them on practical architectures? In the absence of general guidelines of this type, it is useful to look in detail at a given practical problem to see how to design an optimal hardware system to implement it: this is done in the next section.

26.6 OPTIMAL IMPLEMENTATION OF IMAGE ANALYSIS ALGORITHMS

The particular algorithm considered in this section was examined earlier by Davies and Johnstone (1986, 1989). It involves the inspection of round food products (see also Chapter 20). The purpose of the analysis is to show how to make a systematic selection between available hardware modules (including computers), so that it can be guaranteed that the final hardware configuration is optimal in specific ways and in particular with regard to relevant cost−speed tradeoffs.

In the particular food product application considered, biscuits are moving at rates of up to 20 per second along a conveyor. Since the conveyor is moving continuously, it is natural to use a line-scan camera to obtain images of the products. Before they can be scrutinized for defects and their sizes measured, they have to be located accurately within the images. Since the products are approximately circular, it is straightforward to employ the Hough transform technique for the purpose (see Chapter 12): it is also appropriate to use the radial intensity histogram approach to help with the task of product scrutiny (Chapter 20). In addition, simple thresholding can be used to measure the amount of chocolate cover and

Table 26.1 Breakdown of the Inspection Algorithm

	Function	Description	Time (s)	Cost (£)	*c/t* (£/ms)
1.	Acquire image	1 × 1	—	1000	—
2.	Clear parameter space	1 × 1	0.017	200	11.8
3.	Find edge points	3 × 3	4.265	3000	0.7
4.	Accumulate points in parameter space	1 × 1	0.086	2000	23.3
5.	Find averaged center	—	0.020	2000	100.0
6.	Find area of product	1 × 1	0.011	100	9.1
7.	Find light area (no chocolate cover)	1 × 1	0.019	200	10.5
8.	Find dark area (slant on product)	1 × 1	0.021	200	9.5
9.	Compute radial intensity histogram	1 × 1	0.007	400	57.1
10.	Compute radial histogram correlation	1-D	0.013	400	30.8
11.	Overheads for functions 6–10	—	0.415	1200	2.9
12.	Calculate product radius	1-D	0.047	4000	85.1
13.	Track parameters and log	—	0.037	4000	108.1
14.	Decide if rejection is warranted	—	0.002	4000	2000.0
	Time for whole algorithm		4.960		

Source: © *IMechE 1986.*

certain other product features. The main procedures in the algorithm are summarized in Table 26.1. Note that the Hough transform approach requires the rapid and accurate location of edge pixels, which is achieved using the Sobel operator and thresholding the resulting edge enhanced image. Edge detection is in fact the only 3 × 3 neighborhood operation in the algorithm and hence it is relatively time consuming; rather less processing is required by the 1 × 1 neighborhood operations (Table 26.1); then come various 1-D processes such as analysis of radial histograms. The fastest operations are those such as logging variables, which are neither 1-D nor 2-D processes.

26.6.1 Hardware Specification and Design

On finalization of the algorithm strategy, the overall execution time was found to be about 5 s per product (Davies and Johnstone, 1986).[3] With product flow rates of the order of 20 per second, and software optimization subject to severely

[3]Although the particular example is dated, the principles involved are still relevant and worth following here.

diminishing returns, a further gain in speed by a factor of around 100 could be obtained only by using special electronic hardware. In this application, a compromise was sought with a single CPU linked to a set of suitable hardware accelerators. In this case, the latter would have had to be designed specially for the purpose, and indeed some were produced in this way. However, in the discussion below, it is immaterial whether the hardware accelerators are made specially or purchased—the object of the discussion is to present rigorous means for deciding which software functions should be replaced by hardware modules.

As a prerequisite to the selection procedure, Table 26.1 lists execution times and hardware implementation costs of all the algorithm functions: the figures are somewhat notional since it is difficult to divide the algorithm rigorously into completely independent sections. However, they are sufficiently accurate to form the basis for useful decisions on the cost-effectiveness of hardware. Since the aim is to examine the principles underlying cost–speed tradeoffs, the figures presented in Table 26.1 are taken as providing a concrete example of the sort of situation that can arise in practice.

26.6.2 Basic Ideas on Optimal Hardware Implementation

The basic strategy that was adopted for deciding on a hardware implementation of the algorithm is as follows:

1. Prioritize the algorithm functions so that it is clear in which order they should be implemented.
2. Find some criterion for deciding when it is *not* worth proceeding to implement further functions in hardware.

From an intuitive point of view, function prioritization seems a simple process: basically functions should be placed in order of cost-effectiveness, i.e., those saving the most execution time per unit cost (when implemented in hardware) should be placed first, and those giving lesser savings should be placed later. Thus, we arrive at the *c/t* (*cost/time*) criterion function. Then with limited expenditure we achieve the maximum saving in execution time, i.e., the maximum speed of operation.

To decide at what stage it is not worth implementing further functions in hardware is arguably more difficult. Excluding here the practical possibility of strict cost or time limits, the ideal solution results in the optimal balance between total cost and total time. Since these parameters are not expressible in the same units, it is necessary to select a criterion function such as $C \times T$ (*total cost* $\times$ *total time*), which, when minimized, allows a suitable balance to be arrived at automatically.

The procedure outlined above is simple and does not take account of hardware that is common to several modules. This "overhead" hardware must be implemented for the first such module and is then available at zero cost for subsequent modules. In many cases, a speed advantage results from the use of overhead hardware. In the example system, it is found that significant economies are possible when implementing functions 6–10, since common pixel scanning circuitry may be used. In addition, note that any of these functions that are *not* implemented in hardware engender a time overhead in software. This, and the fact that the time overhead is much greater than the sum of the software times for functions 6–10, means that once the initial cost overhead has been paid it is proven to be best (in *this* case) to implement all these functions in hardware.

Trying out the design strategy outlined above gives the c/t ratio sequence shown in Table 26.2. A set of overall times and costs resulting from implementing in hardware all functions down to and including the one indicated is now deduced. Examination of the column of $C \times T$ products then shows where the tradeoff between hardware and software is optimized: this occurs here when the first 13 functions are implemented in hardware.

The analysis presented in this section clearly gives only a general indication of the required hardware–software tradeoff. Indeed, minimizing $C \times T$ indicates an overall "bargain package," whereas in practice the system might well have to meet certain cost or speed limits. In this food product application, it was necessary to aim at an overall cost of less than £10,000. By implementing functions 1, 3 and 6–11 in hardware, it was found possible to get to within a factor 3.6 of the optimal $C \times T$ product. Interestingly, using an upgraded host processor it proved possible to get within a much smaller factor (1.8) of the optimal tradeoff, with the same number of functions implemented in hardware (Table 26.2): indeed, it is a particular advantage of using this criterion function approach that the choice of which processor to use becomes automatic.

The paper by Davies and Johnstone (1989) goes into some depth concerning the choice of criterion function, showing that more general functions are available and there is a useful overriding geometrical interpretation—that global concavities are

Table 26.2 Speed–Cost Tradeoff Figures

Function (see Table 26.1)	c/t (£/ms)	t (s)	c (£)	T (s)	C (£)	$C \times T$ (£–s)	$C' \times T'$ (£–s)
—		—	6000	4.990	6000	29,940	15,080
3	0.7	4.265	3000	0.725	9000	6530	3190
6–11	5.1	0.486	2500	0.239	11,500	2750	1400
2	11.8	0.017	200	0.222	11,700	2600	1350
4	23.3	0.086	2000	0.136	13,700	1860	1040
12	85.1	0.047	4000	0.089	17,700	1580	1010
5	100.0	0.020	2000	0.069	19,700	1360	860
13	108.1	0.037	4000	0.032	23,700	760	770
14	2000.0	0.002	4000	0.030	27,700	830	860

In this table, the first entry corresponds to the base system cost, including computer, camera, frame store, backplane, and power supply. The other entries are derived from Table 26.1 by ordering the c/t values (see text). The final column shows the figures obtained using an upgraded host processor.
Source: © IMechE 1986.

being sought on the chosen curve in $f(C)$, $g(T)$ space. The paper also places the problem of overheads and relevant functional partitions on a more rigorous basis. Finally, a later paper (Davies et al., 1995) emphasizes the value of software solutions, achieved, e.g., with the aid of arrays of digital signal processing (DSP) chips.

26.7 SOME USEFUL REAL-TIME HARDWARE OPTIONS

In the 1980s, real-time inspection systems typically had many circuit boards containing hundreds of dedicated logic chips, although some of the functionality was often implemented in software on the host processor. This period was also a field-day for the Transputer type of microprocessor, which had the capability for straightforward coupling of processors to make parallel processing systems. In addition, the "bit-slice" type of microprocessor permitted easy expansion to larger word sizes, although the technology was also capable of use in multi-pixel parallel processors. Finally, the VLSI type of logic chip was felt by many to present a route forward, particularly for low-level image processing functions.

In spite of all these competing lines, what gradually emerged in the late 1990s as the predominant real-time implementation device was the digital signal processing (DSP) chip: this had evolved earlier in response to the need for fast 1-D signal processors, capable of performing such functions as Fast Fourier Transforms and processing of speech signals. The reason for its success lays in its convenience and programmability (and thus flexibility) and its high speed of operation. By coupling DSP chips together it was found that 2-D image processing operations could be performed both rapidly and flexibly, thereby to a large extent eliminating the need for dedicated random logic boards, and at the same time ousting Transputers and bit-slices.

However, over the same period, single-chip field-programmable logic arrays (FPGAs) were becoming more popular and considerably more powerful, whereas, in the 2000s, microprocessors are being embedded on the same chip. At this stage, the concept is altogether more serious and in the 2000s one has to think quite carefully to obtain the best balance between DSP and FPGA chips for implementing practical vision systems.

In fact, another contender in this race is the "ordinary" PC. Although in the 1990s, this had not normally been regarded as a suitable implementation vehicle for real-time vision, the possibility of implementing some of the slower real-time functions using a PC gradually arose though the relentless progression of Moore's Law, whereas other work on special software designs (see Chapter 21) showed how this line of development could be extended to faster running applications. In fact, we are now at the exciting stage that a single unaided PC with an embedded operating system is sufficient to run a proportion of machine vision applications—a proportion that is expected to grow substantially in the coming decade. Furthermore, we can anticipate that, over time, the whole emphasis of real-time vision will move away from speed being the dominating influence. At that stage effectiveness, accuracy, robustness, and reliability (what one might call "fitness for purpose") will be all that matters: for the first time we will be free to design ideal vision systems. The main word of caution is that this ideal will not necessarily apply for all possible applications—one can imagine exceptions, e.g., where ultrahigh speed aircraft have to be controlled or where huge image databases have to be searched rapidly.

Overall, we can see a progression amid all the hardware developments outlined above: this is a move from random logic design to the use of software-based PEs, and further, one where the software runs not on special devices with limited capability but on conventional computers for which (a) very long instruction words are not needed, (b) machine code or assembly language programming is not necessary to get the most out of the system (so standard languages can be used), and (c) overt parallel processing (beyond that available in the central processing chip of a standard PC) is no longer crucial.[4] The advantages in terms of flexibility are dramatic compared with the early days of machine vision.

The trends described above are underlined by the publications discussed in Section 26.12, and are summarized in Table 26.3.

26.8 SYSTEMS DESIGN CONSIDERATIONS

Having focussed on the problems of real-time hardware design, it is now necessary to get a clearer idea of the overall systems design process. In fact, one of the most important limitations on the rate at which machine vision systems can be produced is the lack of flexibility of existing design strategies: this applies especially to inspection systems. To some extent, this problem stems from lack of understanding of the basic principles of vision upon which inspection systems might optimally be based. In addition, there is the problem of lack of knowledge of what goes into the design process. It is difficult enough designing a complete inspection system, including all the effort that goes into producing a cost-effective real-time hardware implementation, without having to worry at the same time whether the schema used is generic or adaptable to other products. Yet this is a crucial factor that deserves a lot of attention.

An important factor impeding progress in this area is lack of detailed information on commercial systems, and lack of space in published papers: in the latter case what suffers is "know-how"—particularly on creativity aspects (journals see their role as promoting scientific methodology and results rather than subjective design notions). We explore the situation in more detail in the following section.

[4]Nevertheless, some functions such as image acquisition and control of mechanical devices will have to be carried out in parallel to prevent data bottlenecks and other holdups.

Table 26.3 Hardware Devices for the Implementation of Vision Algorithms

Device	Function	Summary of Properties
PC	*Personal computer or more powerful workstation*: complete computer with RAM, hard disk and other peripheral devices. Would need an embedded (restricted) operating system in a real-time application.	• Fast • Medium cost • Extremely flexible • Should be envisioned as a software device
MP	*Microprocessor*: single chip device containing CPU + cache RAM. The core element of a PC.	• Fast • Low cost • Extremely flexible • Should be envisioned as a software device
DSP	*Digital signal processor*: long instruction word MP chip designed specifically for signal processing—high processing speed on a restricted architecture.	• Very fast • Low cost • Highly flexible (some flexibility sacrificed for speed) • Should be envisioned as a software device
FPGA	*Field programmable gate array*: random logic gate array with programmable linkages; may even be dynamically reprogrammable within the application. The latest devices have flip-flops and higher level functions already made up on chip, ready for linking in; some such devices even have one or more MPs on board.	• Fast • Low-to-medium cost • Extremely flexible • Should be envisioned as a hardware device, commonly slaved to a DSP • Can be a software device if controlled by on-chip MPs
LUT	*Lookup table*: RAM or ROM. Useful for fast lookup of crucial functions. Normally slave to a MP or DSP.	• Very fast • Low cost • Extremely flexible, if built using RAM • Should be envisioned as a slave software device
ASIC	*Application-specific integrated circuit*: contains devices such as Fourier transforms, or a variety of specific SP or vision functions. Normally slave to a MP or DSP.	• Very fast • Medium cost • Inflexible (flexibility sacrificed for speed) • Should be envisioned as a slave software device
Vision chip	*Vision chips are ASICs that are devised specifically for vision*: they may contain several important vision functions, such as edge detectors, thinning algorithms, and	• Very fast • Medium cost • Inflexible (flexibility sacrificed for speed)

(*Continued*)

Table 26.3 (Continued)

Device	Function	Summary of Properties
	connected components analyzers. Normally slave to a MP or DSP.	• Should be envisioned as a slave software device
VLSI	*Custom VLSI chip*: this commonly has many components from gate level upwards frozen into a fixed circuit with a particular functional application in mind. (Note, however, that the generic name includes MPs, DSPs, although we shall ignore this possibility here.)	• Fast • High cost • Inflexible • Should be envisioned as a hardware device • Normally slave to a MP or DSP
GPU	Graphics processing unit on a PC or other workstation. Although designed for computer games and other graphics applications, GPUs are able to provide valuable functionality for computer vision.	• Very fast • Substantial power requirements

The only high-cost item is the VLSI chip*: the cost of producing the masks is only justifiable for high-volume products, such as those used in digital TV. In general, high cost means more than £10,000, medium cost means around £2000, and low cost means less than £100.*

If there is a single winner in this table from the point of view of real-time applications, it is the FPGA, supposing only that it contains sufficient onboard raw computing power to make the optimum use of its available random logic. Its dynamic reprogrammability is potentially extremely powerful, but it needs to be known how to make best use of it. It has been usual to slave FPGAs to DSPs or MPs,[a] *but the picture changes radically for FPGAs containing on-chip MPs.*

For a discussion of GPUs, see Section 26.12.3.

[a]*In fact, the FPGA and the DSP complement each other exceptionally well, and it has been common practice to use them in tandem.*

26.9 DESIGN OF INSPECTION SYSTEMS—THE STATUS QUO

As practiced hitherto, the design of an inspection system has a number of stages, much as in the list presented in Table 26.4. While this list is clearly incomplete (e.g., it includes no mention of lighting systems), it is a useful start, and does reveal something about the creativity aspects—by admitting that reassessments of the efficacy of algorithms may be needed. In fact, there are necessarily one or two feedback loops, through which the efficacy can be improved systematically, again and again, until operation is adequate, or indeed until the process is abandoned.[5] The underlying “process” appears to be:

[5]It is in the nature of things that you can’t be totally sure whether a venture will be a success without trying it. Furthermore, in the hard world of industrial survival, part of adequacy means producing a working system and part means making a profit out of it. This section must be read in this light.

Table 26.4 Stages in the Design of a Typical Inspection System

1.	Hearing about the problem
2.	Analyzing the situation
3.	Looking at the data
4.	Testing obvious algorithms
5.	Realizing limitations
6.	Developing algorithms further
7.	Finding things are difficult and to some extent impossible
8.	Doing theory to find the source of any limitations
9.	Doing further tests
10.	Getting an improved approach
11.	Reassessing the specification
12.	Deciding whether to go ahead
13.	Completing a software system
14.	Assessing the speed limitations
15.	Starting again if necessary
16.	Speeding up the software
17.	Reaching a reasonable situation
18.	Putting through 1000 images
19.	Designing a hardware implementation
20.	Revamping the software if necessary
21.	Putting through another 100,000 images
22.	Assessing difficulties regarding rare events
23.	Assessing timing problems
24.	Validating the final system

Source: © *IEE 1997.*

```
create a basic scheme;
do
    improve current scheme;
    if time up then stop;
    if no further ideas then stop;
until an adequate system is obtained;
```

The "if no further ideas" clause can be fulfilled if no way is found of making the system fast enough or low enough in cost, or of high enough specification. This would account for most contingencies that could arise.

The problem with the above process is that it represents *ad hoc* rather than scientific development, and there is no guarantee that the solution that is reached is optimal. Indeed, specification of the problem is not insisted upon (except to the level of "adequacy"), and if specification and aims are absent, it is impossible to

Table 26.5 Complexities of the Design Process

• It is not always evident that there is a solution, or at least a cost-effective one.
• Specifications cannot always be made in a nonfuzzy manner.
• There is often no rigorous scientific design procedure to get from specification to solution (there is certainly no guaranteed way of achieving this optimally).
• The optimization parameters are not obvious; nor are their relative priorities clear.
• It can be quite difficult to discern whether one solution is better than another.
• Some inspection environments make it difficult to tell whether a solution is valid or not.
Source: © *IEE 1997.*

judge whether the success that is obtained is optimal or not. In an engineering environment we ought to be insisting on problem specification first, solution second. However, things are more complex than this—as will be seen from Table 26.5.

In the last case in Table 26.5, there are some types of fault[6] that are particularly rare, so that only one may arise in many million cases, or, equivalently, every few weeks or months. Thus there is little statistical basis for making judgments of the risk of failure, and no proper means of training the system so that it can learn to discriminate these faults. For these sorts of reasons, making a rigorous specification and systematically trying to meet it are extremely difficult, though it is still worth trying to do so.

Let us return to the first of the complexities listed in Table 26.5. Although in principle it is difficult to know if there is a solution to a problem, nevertheless it is frequently possible to examine the computer image data that arise in the application, and see whether the eye can detect the faults or the foreign objects. If it can, this represents a significant step forward, as it means that it should be possible to devise a computer algorithm to do the same thing. What will then be in question will be whether we are creative enough to design such an algorithm, and to ensure that it is sufficiently rapid and cost-effective to be useful.

A key factor at this point is making an appropriate choice of design strategy: with structured types of image data, for example, we can ask the following:

1. Should boundary tracking be employed?
2. Should Hough transform line finding be used?
3. Should corner detection be used?

Which of these alternatives could, *with the particular dataset*, lead to algorithms of appropriate speed and robustness? On the other hand, for data that is fuzzy or

[6]An important class of rare faults is that of foreign objects, including the hard and sometimes soft contaminants targeted by X-ray inspection systems.

where objects are ill-defined, would artificial neural networks be more useful than conventional programming? Overall, the types of data, *and* the types of noise and background clutter that accompany it, are key to the final choice of algorithm.

26.10 SYSTEM OPTIMIZATION

To proceed further, we need to examine the optimization parameters that are relevant. In fact, there are arguably rather few of these:

1. Sensitivity
2. Accuracy
3. Robustness
4. Adaptability
5. Reliability
6. Speed
7. Cost

Of course, these one-word parameters are somewhat imprecise. For example, "reliability" can mean a multitude of things, including freedom from mechanical failures such as might arise with the camera being shaken loose, or the illumination failing; or where timing problems occur when additional image clutter results in excessive analysis time, so that the computer can no longer keep up with the real-time flow of product. Be this as it may, several of the parameters have been shown to depend on each other—e.g., sensitivity and accuracy, cost, and speed: more will be said about this in Chapter 27. Thus, we are working in a multidimensional space with various constraining curves and surfaces. In the worst case, all the parameters will be interlinked, corresponding to a single constraining surface, so that adjusting one parameter forces adjustment of at least one other. There is also the possibility that the constraining surface will impose hard limits on the values of some parameters.

In fact, each algorithm will have its own constraining surface which will in general be separate from that of others. Placing all such surfaces together will create some sort of envelope surface, corresponding to the limits of what is possible with *currently available* algorithms (Fig. 26.3). There will also be an envelope surface, corresponding to what is possible with *all possible* algorithms, i.e., including those that have not yet been developed. Thus there are limits imposed by creativity and ingenuity, and it is not known at what stage such limits might be overcome.

Returning to the constraining surface, this can be seen to provide an element of choice in the situation. Do we prefer a sensitive algorithm or a robust one, a reliable algorithm or a fast one? And so on. However, algorithms are rarely accurately describable as robust or nonrobust, reliable or nonreliable, and the multiparameter space concept with its envelope constraining surfaces clearly allows for variations in such parameters. But at the present stage of development of the

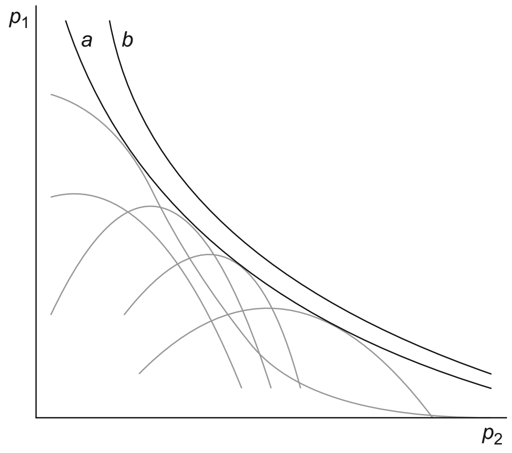

FIGURE 26.3

Optimization curves for a two-parameter system. The gray curves result from individual algorithms. (a) The envelope of the curves for all known algorithms. (b) The limiting curve for all possible algorithms.

Source: © *World Scientific 2000*

subject, the problem we have is that these constraining surfaces are rarely known; in addition, the algorithms that could provide access to their ideal forms are largely unknown: even the algorithms that are available are of largely unmapped capabilities. It is also the case that means are not generally known for selecting optimal working points (an interesting exception is that elucidated by Davies and Johnstone (1989) relating to optimization of cost/speed tradeoffs for image-processing hardware). Thus, there is a long way to go before the algorithm design and selection process can be made fully scientific.

26.11 CONCLUDING REMARKS

This chapter has studied the means available for implementing image analysis algorithms in real time. A number of sequential and parallel processing architectures have been considered, as well as other approaches involving use of DSP and FPGA chips or PCs with embedded operating systems. In addition, means of selecting between the various realizable schemes have been studied, these being based on criterion function optimizations.

The field is characterized on the one hand by elegant parallel architectures that would cost an excessive amount in most applications, and on the other by hardware solutions that are optimized to the application yet which are bound to lack generality. In fact, it is easy to gain too rosy a view of the elegant parallel architectures that are available: their power and generality could easily make them an overkill in dedicated applications, while reductions in generality could mean that their PEs spend much of their time waiting for data or performing null operations. Overall, the subject of image analysis is quite variegated and it is difficult to find a set of "typical" algorithms for which an "average" architecture can be designed whose PEs will always be kept usefully busy. Thus, the subject of mapping algorithms to architectures—or, better, of matching algorithms *and* architectures, i.e., designing them together as a system—is a truly complex one for which obvious solutions turn out to be less efficient in reality than when initially envisioned. Perhaps the main problem lies in envisaging the nature of image analysis, which turns out to be a far more abstract process than image processing (i.e., image-to-image transformation) ideas would indicate.

Interestingly, the problems of envisaging the nature of image analysis and of matching hardware to algorithms are gradually being bypassed by the relentless advance in the speed and power of everyday computers: we have seen that this means that there will be less need for special dedicated hardware for real-time implementation, especially if suitable algorithms such as those outlined in Section 21.4 can be developed: all this represents a very welcome factor for those developing vision systems for industry.

One topic which has been omitted for reasons of space is that of carrying out sufficiently rigorous timing analysis of image analysis and related tasks so that the overall process is *guaranteed* to run in real time. This is especially important in the case of inspection. In this context, a particular problem is that unusual image data conditions could arise which might engender additional processing, thereby setting the system behind schedule: this situation could arise because of excessive noise, because more than the usual number of products appear on a conveyor, because the heuristics in a search tree prove inadequate in some situation, or for a variety of other reasons. One way of eliminating this type of problem is to include a watchdog timer, so that the process is curtailed at a particular stage (with the product under inspection being rejected, or other suitable action being taken). However, this type of solution is crude, and sophisticated timing analysis methods now exist, based on Quirk analysis: Môtus and Rodd are protagonists of this type of rigorous approach. The reader is referred to Thomas et al. (1995) for further details and to Môtus and Rodd (1994) for an in-depth study.

Finally, the last few sections have aimed at highlighting certain weaknesses in the system design process and indicating how the subject can be developed further for the benefit of industry. Perusal of the earlier chapters will quickly show that we can now achieve a lot even though our global design base is limited. The key to our ability to solve the problems, and to know that they are solvable, lies in the capability of the human visual system for carrying out relevant visual tasks. Note, however, that there is sound reason for replacing the human eye for these visual tasks—so that (for example) 100% tireless inspection can be carried out and so that it can be achieved consistently and reliably to known standards. Clearly, the same sentiments apply for many other applications, such as driver assistance, surveillance for crime detection, and so on (see, for example, Chapters 22 and 23).

> The greatest proportion of the real-time vision system design effort used to be devoted to the development of dedicated hardware accelerators. This chapter has underlined that this problem is gradually receding, and that it is already possible to design reconfigurable FPGA systems that in many cases bypass the need for special parallel processor systems.

26.12 BIBLIOGRAPHICAL AND HISTORICAL NOTES[7]

26.12.1 General Background

The problem of implementing vision in real time presents exceptional difficulties, since in many cases megabytes of data have to be handled in times of the order of hundredths of a second, and often repeated scrutiny of the data is necessary. Hundreds of architectures have been considered for this purpose—most having some degree of parallelism. Of these, many have highly structured forms of parallelism, such as SIMD machines. The idea of a SIMD machine for image parallel operations goes back to Unger (1958), and strongly influenced later work both in the UK and in the USA. For an overview of early SIMD machines, see Fountain (1987).

The SIMD concept was generalized by Flynn (1972) in his well-known classification of computing machines (Section 26.5). Subsequently, there was a great proliferation of interconnection networks (Feng, 1981): Reeves (1984) reviewed a number that would be useful for image processing. An important problem is that of performing an optimal mapping between algorithms and architectures but for this to be possible a classification of parallel algorithms is needed to complement that of architectures: some such work was undertaken by Kung (1980), Cantoni and Levialdi (1983), and Chiang and Fu (1983).

For one period in the 1980s it became an important challenge to design VLSI chips for image processing and analysis. Offen (1985) and Fountain (1987) summarized many of the possibilities. The problems targeted in that era were defining the most useful image processing functions to encapsulate in VLSI and to understand how best to partition the algorithms taking account of chip limitations. At a

[7]Because of the rapid aging processes that computer and hardware development are subject to, it has been found necessary to divide this section into three parts. However, in spite of the obvious temporal development between these parts, the reader is recommended not to rush hastily past the first two of them, as they embody important principles, ideas and methods that are still highly relevant today.

more down-to-earth level, many image processing and analysis systems did not employ sophisticated parallel architectures but “merely” very fast serial processing techniques with hardwired functions—often capable of processing at video rates (i.e., giving results within one TV frame time).

Other solutions included the use of bit-slice types of device (Edmonds and Davies, 1991), and DSP chips (Davies et al., 1995), while special purpose multiprocessor designs were described for implementing multiresolution and other Hough transforms (Atiquzzaman, 1994).

26.12.2 Developments Since 2000

As indicated in Section 26.7, VLSI solutions to the production of rapid hardware for real-time applications have gradually given way to the much more flexible software solutions permitted by DSP chips and to the highly flexible FPGA type of system which permits random logic to be implemented with relative ease. In fact, FPGAs offer the possibility of dynamic reconfigurability, which may ultimately be very useful for space probes and the like, though it may prove to be an unnecessary design burden for most vision applications.[8] In this respect it is interesting to note the trend toward hybrid CPU/FPGA chips (Andrews et al., 2004) that will have optimal combinations of software and random logic available awaiting software and hardware programming (see also Batlle et al., 2002).

Be this as it may, in the early 2000s considerable attention was still focussed on VLSI solutions to vision applications (Tzionas, 2000; Mémin and Risset, 2001; Wiehler et al., 2001; Urriza et al., 2001), and it is clear that there are bound to be high-volume applications (such as digital television) where this will remain the best approach. Similarly, a lot of attention is still focussed on SIMD and linear array processor schemes, as there will always be facilities that offer ultrahigh-speed solutions of this type, and in any case small-scale implementations are likely to be cheap and usable for those (mainly low-level vision) applications that match this type of architecture. Examples of this appear in the papers by Hufnagl and Uhl (2000), Ouerhani and Hügli (2003), and Rabah et al. (2003).

Moving on to FPGA solutions, we find these embodying several types of parallelism and applied to underwater vision applications and robotics (Batlle et al., 2002), subpixel edge detection for inspection (Hussmann and Ho, 2003), and general low-level vision applications, including those implementable as morphological operators (Draper et al., 2003).

DSP solutions, some of which also involve FPGAs, include those by Meribout et al. (2002) and Aziz et al. (2003). It is useful to make comparisons with the Datacube MaxPCI (containing a pipeline of convolvers, histogrammers and other devices) and other commercially available boards and systems—see Broggi et al.

[8]However, see Kessal et al. (2003) for a highly interesting investigation of the possibilities—in particular showing that dynamic reconfigurability of a real-time vision system can already be achieved in milliseconds.

(2000a, 2000b), Yang et al. (2002), and Marino et al. (2001). This last paper employs a cleverly conceived architecture incorporating extensive use of lookup tables to perform high-speed matching functions that lead to road following: in fact, the matching functions involve location of interest points followed by high-speed search for matching them between corresponding blocks of adjacent frames. An interesting feature is the use of a residue number system to (effectively) factorize large lookup tables into several much smaller and more manageable lookup tables.

26.12.3 More Recent Developments

By 2011, hardware design for fast vision implementations has moved forward another stage, and various types of embedded real-time systems are vying with each other for supremacy—ASICs, DSPs, FPGAs, and GPUs (graphics processing units) being the main contenders. Ambrosch and Kubinger (2010) have developed stereo algorithms suitable for implementation in both ASIC and FPGA forms. However, as ASICs offer higher performance but also higher costs, these authors decided it would be more realistic to use FPGAs at least for prototyping and testing. Appiah et al. (2010) made a similar decision for their video object segmentation hardware, consisting of a single-chip FPGA together with four blocks of RAM. It embodied algorithms for foreground detection (using multimodal background modeling) and connected components labeling. Various parts of the algorithm run in parallel and the whole system is pipelined for maximum efficiency. While tackling nominally the same stereo imaging problem as Ambrosch and Kubinger (2010), Humenberger et al. (2010) produced three implementations using, respectively, a PC, a GPU, and a DSP. The latter two give real-time performance. However, the GPU is by far the fastest but has the highest power consumption. Interestingly, the DSP gives the most stable performance, with the processing times of successive frames being almost identical—in contrast to the other two implementations, which vary over several percent. In these cases, large data caches and high-level operating systems severely affect the predictability of the worst-case execution times.

The recent introduction of GPUs into the scheme of things is interesting: curiously, it was the games market that led to this way forward, as the demand for realistic computer games operating real-time 3-D HCI (human−computer interaction) necessitated exactly this sort of technology. Here, a relevant question is how dedicated GPUs are to games programs as distinct from operations that can be of value for vision. In fact, any fears on this score would appear to be groundless, as (to take one convenient example) May et al. (2010) have shown that (at minimum) *all* the SIFT modules (see Chapter 6) can be programmed on a general-purpose GPU.

Medeiros et al. (2010) developed algorithms for a parallel histogram-based particle filter for object tracking on SIMD-based smart cameras. The arrays of photoelectric elements in a camera map well to internal SIMD processors, though in practice cost and complexity considerations may, as here, limit the SIMD architecture to linear rather than area arrays. In addition, particle filters lend themselves well to parallel implementation since there are no data dependencies among particles. In fact, the research described in this paper makes much use of histograms, both color histograms and more complex histograms of oriented gradients (HOG), and again these map well to parallelism and to SIMD architectures. With these algorithms and this technology, the authors were able to achieve robust tracking of objects including humans at up to 30 frames per second.

Marzotto et al. (2010) developed a real-time roadway path extraction and tracking system for use on road vehicles, and implemented it on an FPGA platform. The ultimate purpose was that of driver assistance by providing a lane departure warning system. The proposed algorithm was designed to be completely embedded in FPGA hardware and to be capable of processing wide-VGA video sequences at 30 frames per second. The basic algorithm was targeted at locating road lane markings and made use of RANSAC for line and curve fitting. However, the overall algorithm also included a substantial amount of pre-processing in the form of noise reduction, histogram stretching, edge detection, edge thinning, automatic thresholding, and morphological filtering, together with post-processing using a Kalman filter. Nevertheless, all these and other functions were embedded within the FPGA, making full internal use of "DSPs" (programmable multiplier-accumulator units), "BRAMs" (blocks of RAM elements), and "slices" (configurable logic blocks). In fact, the system incorporated 34 DSPs, 32 BRAMs, and 8398 slices, but made use of only about 30% of the FPGA's hardware resources. While the main tests described in the paper involved simulations, preliminary tests on a real vehicle provided performances of up to 60 fps at normal VGA resolution.

Finally, to underline the role and importance of GPUs in this area, the success of the "Kinect" human motion capture system designed by engineers from Microsoft Research and provided in Xbox 360 should be noted: the unit sold 8 million devices within 2 months of its launch in November 2010, making it the fastest selling consumer electronics device in history.