In *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1994.

# Hardware-Software Codesign of Multidimensional Programs

Wayne Luk, Teddy Wu and Ian Page
Programming Research Group,
Oxford University Computing Laboratory,
11 Keble Road, Oxford, England OX1 3QD

**Abstract**

*We present a method for parametrised partitioning of multidimensional programs for acceleration using a hardware coprocessor. The method involves a divide-and-conquer structure, with the "divide" and "merge" phases carried out by a general-purpose processor while the "conquer" phase is handled by application-specific hardware. The partitioning strategy has been captured in a simple functional language, and we have automated the production of partitioned programs in this language. Our approach has been tested on an FPGA-based system using a number of computer vision algorithms, including the Canny edge detector, and the performance is compared against executing the programs on the PC host.*

## 1 Introduction

The objective of our research on hardware-software codesign is to develop systems containing both hardware and software components with higher quality, in shorter time, and at lower cost than existing ones. Recent advances in programmable logic, particularly Field-Programmable Gate Arrays (FPGAs), offer a rapid route for building reconfigurable hardware accelerators. It is clear that a cost-effective implementation can be obtained by combining fast but expensive hardware with slow but cheap microprocessors; in particular, methods for appropriate partitioning of computations for coordinated execution in hardware and in software are essential.

This paper describes a strategy for partitioning programs operating on multidimensional arrays of data. The aim is to convert such programs by correctness-preserving transformations into parametrised descriptions, which can then be used to generate efficient implementations with a variety of hardware-software trade-offs. Our method may be applied to overcome physical limitations – such as having an insufficient amount of memory – in hardware accelerators. It partitions an program into three steps: "divide", "conquer" and "merge". A general-purpose processor, usually proficient in executing relatively complex, data-dependent and variable size operations, can be used for dividing the input data into smaller pieces and for merging the results of processing the pieces. Purpose-built hardware is used for processing the partitioned data, often in a systolic mode [7].

Many algorithms can be expressed clearly and concisely in a functional language; functional programs are often used as executable specifications and in rapid prototyping. Moreover, simple equational transformations can be used to reason about functional programs. Our partitioning strategy has been captured in a functional notation adopted by Bird and Wadler [2]. In this notation, applying $g$ to an argument $x$ – which can itself be a function – will be denoted by $g\ x$; function application is left associative, so $g\ x\ y\ z$ = $((g\ x)\ y)\ z$. The infix operator $\cdot$ denotes function composition: $(f \cdot g)\ x = f\ (g\ x)$. It is the least binding among binary operators, so $f \cdot g\ x = f \cdot (g\ x)$.

It is straightforward to state the correctness requirement for a partitioning strategy in a functional notation. Let $f$ be the program to be partitioned. Our partitioning strategy, given by $partition\ f$, is a composition of three functions $divide$, $conquer$ and $merge$,

$$partition\ f \quad = \quad merge \cdot conquer\ f \cdot divide$$

Later on we shall show that under certain conditions, this definition of $partition$ satisfies the correctness requirement that

$$partition\ f \quad = \quad f$$

In other words, the partitioning strategy does not affect the behaviour of $f$.

Our partitioning strategy has been implemented in a functional programming environment known as Gofer [5], which can be obtained by anonymous ftp from `nebula.cs.yale.edu` in directory `pub/haskell/gofer`. The Gofer language is very similar to the notation that we use, although there are minor syntactic differences. It should not be difficult to implement our method in other functional languages such as Orwell, Miranda or SML.

A number of partitioned programs, including sorting and computer vision routines, have been tested on an FPGA-based hardware accelerator known as CHS2x4 [1]. The CHS2x4 is a full-length IBM AT card which communicates with the host computer through the AT bus. The board consists of three subsystems: the Computation Subsystem which holds eight CAL 1024 chips [6] arranged in a two by four array, the Memory Subsystem which contains 256 kilobytes of SRAM, and the Interface and Control Subsystem which deals with the communication between the board and its host machine. At present data transfer between the CAL chips and the on-board memory is restricted to sequential input and output over a byte-wide channel, controlled by invoking C-library routines provided by the manufacturer. Each CAL chip is an FPGA consisting of 32 by 32 cells. Each cell has a one-bit input port and a one-bit output port on each of its four sides. An input port can be programmed to connect to one or more output ports, or to a function unit which can be programmed to behave either as a two-input combinational logic gate or as a latch.

In the following sections we present the partitioning strategy, and show how it can be adopted in developing multidimensional programs that can be speeded up by the CHS2x4.

## 2 Partitioning Strategy

As explained above, a simple approach has been adopted for partitioning programs: a slow engine is used for arranging data into fixed-size blocks that can be handled by a fast but resource-limited engine. Usually the former corresponds to a general-purpose processor, while the latter is implemented by application-specific hardware. Possible applications of this paradigm include computer vision and medical image processing, where the amount of data is often too large for the hardware coprocessor to handle at once.

Although the “divide” and “merge” phases for some computations involve simply partitioning the input data for the dedicated hardware and concatenating the results, this may not always be the case. For instance, in convolution-based algorithms it is usually necessary to replicate the pixels around the partition boundary in neighbouring blocks. This is taken into account by our partitioning method.

Let $f$ be the function to be partitioned; $f$ is required to satisfy certain conditions to be described later in this paper. To capture hardware constraints for implementing an $n$-dimensional program, we use $n$ constants, $maxp_i$ with $1 \leq i \leq n$, to denote for each dimension the maximum number of elements in a partition. The function $partition$ takes $f$ as argument and returns the partitioned operation. It is defined in terms of three functions, $divide$, $conquer$ and $merge$, each corresponding to one step in the partitioned program:

$$partition\ f \quad = \quad merge \cdot conquer\ f \cdot divide$$

Note that $divide$, $conquer$, $merge$ and the conditions which $f$ must satisfy are specific to the number of dimensions of the program, although the basic principle behind the partitioning strategy is the same for programs of any dimension. Let us begin with the simple case of one-dimensional programs, and later extend the strategy to programs for higher dimensions.

## 3 Partitioning One-Dimensional Programs

A one-dimensional array of data will be represented by a list, the elements of which will be enclosed by square brackets. $\#xs$ denotes the number of elements in list $xs$. List construction is signified by a colon, so

$$\begin{aligned} a : [b, c, d] &= [a, b, c, d] \\ &= a : (b : (c : (d : []))) \end{aligned}$$

The ($+\!\!+$) operator appends two lists, for example

$$[a, b] +\!\!+ [c, d, e] \quad = \quad [a, b, c, d, e]$$

Two functions $take$ and $drop$ will be used in describing our partitioning method. Each of them takes a non-negative integer $n$ and a list $xs$ as arguments. The value of $(take\ n\ xs)$ is the initial segment of $xs$ of length $n$, and the value of $(drop\ n\ xs)$ is the list which remains when the first $n$ elements are removed: so

$$\begin{aligned} take\ 3\,[a, b, c, d, e] &= [a, b, c] \\ drop\ 3\,[a, b, c, d, e] &= [d, e] \end{aligned}$$

Clearly for $0 \leq i \leq \#xs$,

$$take\ i\ xs\ +\!\!+\ drop\ i\ xs \quad = \quad xs$$

During the initial “divide” step, a list of data will be partitioned into a list of shorter lists of equal length, except possibly the last one. For some algorithms such as convolution, it is necessary to replicate the boundary elements; in other words, we need to include in each sub-array a number of elements in neighbouring sub-arrays. The number of replicated elements in successive partitions is given by the constant $margin$, and the maximum number of elements in each partition will be denoted by $maxp$; for $n$-dimensional programs these constants will become $margin_i$ and $maxp_i$ for $1 \leq i \leq n$. Let $maxp' = maxp - margin$: then $divide$ is implemented by the function $divide_1$ in one dimension,

$$\begin{aligned} & divide_1\ xs \\ & \quad = take\ maxp\ xs : (divide_1 \cdot drop\ maxp')\ xs, \\ & \qquad \text{if}\ \ \#xs > maxp \\ & \quad = [xs], \quad \text{otherwise} \end{aligned}$$

This recursive definition indicates that provided that $\#xs > maxp > margin$, the first $maxp$ elements of $xs$ will form the first sublist of the partitioned list, and the remaining sublists can be obtained by applying $divide_1$ again to a version of $xs$ shortened by dropping the first $maxp'$ elements. For instance, given $maxp = 3$ and $margin = 1$, $divide_1\ [0,1,2,3,4,5] = [[0,1,2],[2,3,4],[4,5]]$.

To process each sublist in the list produced by $divide_1$, $conquer$ is given by

$$conquer \quad = \quad map$$

where $map$ applies the function $g$ to each element of a list: $map\ g\ [x_0, x_1, x_2] = [g\ x_0, g\ x_1, g\ x_2]$.

Merging the list of results from $conquer$ is performed by the function $merge_1$ which is defined in terms of a binary operator $\odot$:

$$merge_1 \quad = \quad foldr1\,(\odot)$$

where $foldr1$ is given by

$$\begin{aligned} & foldr1\,(\odot)\ [x_0, x_1, \ldots, x_n] \\ & \quad = x_0 \odot (x_1 \odot \cdots (x_{n-1} \odot x_n) \cdots) \end{aligned}$$

We can now specify what it means for a partitioning strategy to be correct: the same result would be obtained whether or not partitioning is involved. Given the above definitions, it can be shown by induction that

$$f\ xs \quad = \quad partition_1\, f\ xs \tag{1}$$

provided that $\odot$ satisfies

$$\begin{aligned} f\ xs \quad = \quad & (f\ (take\ maxp\ xs)) \\ & \odot (f\ (drop\ maxp'\ xs)) \end{aligned} \tag{2}$$

Since $maxp' = maxp - margin$, Equation 2 shows that if we divide a list into two with $margin$ elements duplicated at the point of division, combining the results using $\odot$ should give the same value as that obtained without partitioning. Different $\odot$'s correspond to different ways of combining the partial results.

A simple example which satisfies Equation 2 is $f = id$, $\odot = +\!\!+$ and $margin = 0$, where $id$ is the identity function given by $id\ x = x$. Another example is $f = sort$, $\odot = mergelist$ and $margin = 0$, where $sort$ sorts a list of elements into ascending order, and $mergelist$ merges two sorted lists to form a sorted list. This method for partitioning sorters has been implemented on the CHS2x4 system [9]. A more complicated example will be considered next.

## 4 One-Dimensional Convolution

Consider first the function $zipWith$, which takes a binary operator $\oplus$ and applies it to corresponding elements of two lists which may have different number of elements:

$$\begin{aligned} & zipWith\ \oplus\ [x_0, x_1, x_2]\ [y_0, y_1, y_2, y_3, y_4] \\ & \quad = \ [x_0 \oplus y_0,\ x_1 \oplus y_1,\ x_2 \oplus y_2] \end{aligned}$$

The length of the result is the same as the length of the shorter of the two arguments. Some readers may recognise that this definition of $zipWith$ is the same as that in Gofer, and is slightly different from $zipwith$ in [2].

One-dimensional convolution can be described by

$$convolve_1\ ws\ xs \quad = \quad map\ g\ [0..(\#xs - \#ws)] \tag{3}$$

where

$$g\ i \quad = \quad (foldr1\,(\oplus)\ .\ zipWith\,(\otimes))\ ws\ (drop\ i\ xs)$$

Note that $[0..m]$ corresponds to the list $[0, 1, \ldots, m]$. An experienced functional programmer may recast $convolve_1$ into a more succinct form using list comprehension [2], but we shall not go into the details here.

The correctness of the definition given by Equation 3 can be demonstrated using symbolic execution. Let us define

$$\begin{aligned} x \oplus y &= \text{“(”} +\!\!+ x +\!\!+ \text{“+”} +\!\!+ y +\!\!+ \text{“)”} \\ x \otimes y &= \text{“(”} +\!\!+ x +\!\!+ \text{“}\times\text{”} +\!\!+ y +\!\!+ \text{“)”} \end{aligned}$$

and

$$\begin{aligned} ws &= [\text{“}w0\text{”},\text{“}w1\text{”},\text{“}w2\text{”}], \\ xs &= [\text{“}x0\text{”},\text{“}x1\text{”},\text{“}x2\text{”},\text{“}x3\text{”},\text{“}x4\text{”},\text{“}x5\text{”},\text{“}x6\text{”},\text{“}x7\text{”}]. \end{aligned}$$

It can then be shown by hand or by the Gofer interpreter that

$$\begin{aligned} & convolve_1\ ws\ xs \\ & \quad = \ [\text{“}((w0 \times x0) + ((w1 \times x1) + (w2 \times x2)))\text{”}, \\ & \qquad \text{“}((w0 \times x1) + ((w1 \times x2) + (w2 \times x3)))\text{”}, \\ & \qquad \text{“}((w0 \times x2) + ((w1 \times x3) + (w2 \times x4)))\text{”}, \\ & \qquad \text{“}((w0 \times x3) + ((w1 \times x4) + (w2 \times x5)))\text{”}, \\ & \qquad \text{“}((w0 \times x4) + ((w1 \times x5) + (w2 \times x6)))\text{”}, \\ & \qquad \text{“}((w0 \times x5) + ((w1 \times x6) + (w2 \times x7)))\text{”}] \end{aligned}$$

The above is an example of symbolic simulation. The corresponding numerical evaluation can be obtained by replacing symbolic operations and symbolic data by numerical ones.

From the properties of functions such as $take$, $drop$ and $zipWith$, it can be shown that $convolve_1\ ws$ satisfies Equation 2 with $\odot = +\!\!+$ and $margin = (\#ws) - 1$. In other words,

$$\begin{aligned} & convolve_1\ ws\ xs \\ & \quad = convolve_1\ ws\ (take\ maxp\ xs) \\ & \qquad +\!\!+\ convolve_1\ ws\ (drop\ maxp'\ xs) \end{aligned}$$

Hence by Equation 1,

$$convolve_1\ ws\ xs \quad = \quad partition_1\ (convolve_1\ ws)\ xs.$$

## 5 Partitioning Multidimensional Programs

An $n$-dimensional array will be represented by a list with $n$ nested levels. Given that $g^0 = id$ and $g^i = g \cdot g \cdot \ldots \cdot g$ ($i$ times) and $hd\ (x : xs) = x$, the definitions of $divide$, $conquer$ and $merge$ can be generalised to deal with nested lists:

$$\begin{aligned} & divide \\ & \quad = map^{n-1}\ divide_n \cdot \ldots \cdot map\ divide_2 \cdot divide_1 \end{aligned}$$

$$\begin{aligned} & conquer \\ & \quad = map^n \end{aligned}$$

$$\begin{aligned} & merge \\ & \quad = merge_1 \cdot map\ merge_2 \cdot \ldots \cdot map^{n-1}\ merge_n \end{aligned}$$

Also for $1 \leq i \leq n$, let $maxp'_i = maxp_i - margin_i$, and

$$\begin{aligned} & divide_i\ xs \\ & \quad = map^{i-1}\ (take\ maxp_i)\ xs \\ & \qquad : (divide_i \cdot map^{i-1}\ (drop\ maxp'_i))\ xs, \\ & \qquad\quad \text{if}\ \ \#(hd^{i-1}\ xs) > maxp_i \\ & \quad = [xs],\ \text{otherwise} \end{aligned}$$

and there exist $n$ binary operators $\odot_1$, $\odot_2$, .., $\odot_n$ such that for $1 \leq i \leq n$,

$$\begin{aligned} & merge_i \\ & \quad = foldr1\ \underbrace{(zipWith\ (zipWith \ldots (zipWith}_{i-1\ terms}\ (\odot_i)) \ldots)) \end{aligned}$$

Given that $f$ is the program to be partitioned, then these binary operators should satisfy

$$\begin{aligned} f\ xs \quad = \quad & \underbrace{(zipWith\ (zipWith \ldots (zipWith}_{i-1\ terms}\ (\odot_i)) \ldots)) \\ & (f\ (map^i\ (take\ maxp_i)\ xs)) \\ & (f\ (map^i\ (drop\ maxp'_i)\ xs)) \qquad (4) \end{aligned}$$

for $1 \leq i \leq n$.

A Gofer program has been written which can generate the appropriate Gofer definitions of $divide$, $conquer$ and $merge$ for a given number of dimensions. For instance, the following definitions are produced for $n = 3$:

$$\begin{aligned} & divide \\ & \quad = (map \cdot map)\ divide_3 \cdot map\ divide_2 \cdot divide_1 \end{aligned}$$

$$\begin{aligned} & conquer \\ & \quad = map \cdot map \cdot map \end{aligned}$$

$$\begin{aligned} & merge \\ & \quad = merge_1 \cdot map\ merge_2 \cdot (map \cdot map)\ merge_3 \end{aligned}$$

The definitions of $divide_1$ and $merge_1$ have been given in a Section 3. The definitions of $divide_2$ and $merge_2$ are similar to those of $divide_3$ and $merge_3$, and will not be included below.

$$\begin{aligned} & divide_3\ xs \\ & \quad = (map \cdot map)\ (take\ maxp_3)\ xs \\ & \qquad : (divide_3 \cdot (map \cdot map)\ (drop\ maxp'_3))\ xs, \\ & \qquad\quad \text{if}\ \ \#((hd \cdot hd)\ xs) > maxp_3 \\ & \quad = [xs],\ \text{otherwise} \end{aligned}$$

$$\begin{aligned} & merge_3 \\ & \quad = foldr1\ (zipWith\ (zipWith\ (\odot_3))) \end{aligned}$$

The next section outlines the application of our partitioning strategy to two-dimensional programs for processing images on computer.

## 6 Implementing Partitioned Programs

Many hardware accelerators have fast local memories, but sometimes these memories may not be large enough to hold all the data. For instance, the memory available on the basic CHS2x4 system restricts the image size to around 19600 pixels in our implementation of the Sobel edge detector (see Section 7). This limitation can be overcome by our partitioning strategy.

To obtain an efficient implementation and to be able to use the libraries provided by the manufacturer, the partitioning strategy for two-dimensional arrays is implemented in C. In order to assist converting a program in C to its partitioned version, a C function `partition` has been developed. The following fragments of an imperative program were produced by hand, and we are currently exploring automated methods for generating them from descriptions in the functional notation adopted in the preceding sections.

Since C allows global variables, in our implementation we pass the position of the top left corner and the size of the sub-images as parameters in function calls instead of passing sub-images as parameters. This method avoids copying sub-images and increases efficiency, since dividing an image into sub-images becomes calculations of the positions of the top left corner and the size of sub-images. It follows that the data structure for an image may include the following fields (Figure 1): `top` and `left`, the $x$ and $y$ co-ordinates of the top left-hand corner of the sub-image; `ht` and `wd`, the height and width of the sub-image; and `buf`, the memory location holding the first pixel of the image. Other fields that are required for a particular program and or for a specific image storage format, such as the number of pixels in an image, may also be included in the data structure.

```
typedef struct
{ int           top, left;
  int           ht, wd;
  pixel *_huge  *buf;
} image;
```

**Figure 1** Data structure for storing images.

The function `partition` requires six parameters: `process`, the name of the function (corresponds to $f$ in the preceding section) for processing the image; `im`, the image to be processed; `max_ht`, the maximum height of a sub-image; `max_wd`, the maximum width of a sub-image; and `margin_ht` and `margin_wd`, which correspond to $margin_1$ and $margin_2$ in the preceding section.

Calculating the positions of the top left corner, the width and the height of sub-images can be performed by a nested for-loop (Figure 2); although there are some optimisations, on the whole the code is written for clarity rather than for efficiency. Since the three stages of the partitioning strategy will be executed sequentially on the CHS2x4, it is more efficient to include the *conquer* step in the same nested loop as *divide*, so the line "`(*process) (&subim);`" corresponds to *conquer* $f$. This applies the function `process` to each sub-image. Merging is included in `process`, because different programs may require different ways of merging the results. If the merging is performed by $+\!\!+$, partial results can be written into the memory block specified by the top left corner, width and height of the sub-image.

To use `partition` properly, the following procedure is suggested. Starting from a C program that implements the function $f$ to be partitioned as a C function `process`,

1. Find the appropriate values for the parameters `max_ht`, `max_wd`, `margin_ht` and `margin_wd`.

2. Modify the functions for processing an image to operate on the proper sub-images.

```
partition (void
  (*process) (image *),
  image *im, /* pointer to an image */
  int max_ht,    int max_wd,
  int margin_ht, int margin_wd)
{
  image subim;
  int   max_ht' = max_ht - margin_ht;
  int   max_wd' = max_wd - margin_wd;

  subim.buf = im->buf;
  for (subim.top = 0;
       subim.top + max_ht' < im->ht;
       subim.top += max_ht')
  {
    subim.ht = max_ht;
    /* insert code for     */
    /* the inner loop here */
  }
  subim.ht = im->ht - subim.top;
  /* insert code for the   */
  /* inner loop here again */
}
```

The code for the inner loop is given by

```
  for (subim.left = 0;
       subim.left + max_wd' < im->wd;
       subim.left += max_wd')
  {
    subim.wd   = max_wd;
    subim.size = subim.ht * subim.wd;
    (*process) (&subim);
  }
  subim.wd   = im->wd - subim.left;
  subim.size = subim.ht * subim.wd;
  (*process) (&subim);
```

**Figure 2** An example of partitioning in C.

3. Look for a method for combining the results from sub-images. The eligibility of the method can be checked by finding a binary operator $\odot$ for merging the partitioned results in each dimension such that Equation 4 is satisfied.

4. Modify the program to perform the selected merging method.

5. Replace the function call to process the image by a call to `partition`.

To benefit from the use of the hardware coprocessor, the speed loss due to overhead in partitioning should be smaller than the speed gained by using the coprocessor. Moreover, the amount of overhead should not increase faster than the time required for processing the image as the image size increases. We can analyse the conditions for obtaining an efficient partitioning as follows.

Let $N^2$ be the number of pixels of an image. The processing time is at least O($N^2$). Overhead is introduced in each step of the partitioned process. The time for dividing the image is O($N^2$), as can be seen from the for-loops in `partition`.

The amount of overhead in merging the results depends on the merging operator $\odot$. It is obvious that this should be no worse than O($N^2$) for the partitioning strategy to be efficient. The total overhead will then be O($N^2$), which means that it is of the same order as, or shorter than, the time required for processing the image.

Further improvement in speed can be obtained if the divide and merge steps can be executed in parallel with the conquer step. This is not possible with the CHS2x4 system, but one can develop a hardware platform with several banks of local memories accessible by the general-purpose processor, such that the hardware can operate on data in one of the memory banks while the software is dividing the data and combining the results in other memory banks. The optimal partitioning can be obtained when the hardware and the software processes complete the execution in a similar length of time.

## 7 The Sobel Edge Detector

Using the strategy described in previous sections, we have partitioned two programs for detecting edges in images, one by Sobel [7] and the other by Canny [3]. The size of a partition is determined by the amount of memory available in our CHS2x4 system. The result of applying the Sobel edge detector will be given in the rest of this section, while the Canny edge detector will be discussed later.

The Sobel edge detector involves a two-dimensional convolution using the masks

$$[[-1,-2,-1],[0,0,0],[1,2,1]]$$

and

$$[[-1,0,1],[-2,0,2],[-1,0,1]]$$

to produce the image gradient in the horizontal and in the vertical direction. The squares of the two gradients are then summed together and compared against a threshold. Our implementation consists of adders, subtractors, registers, multiplexers, magnitude extractors and a multiplier.

Table 1 compares the Sobel edge detector in software on a 386-based PC against the FPGA-assisted version. The stages *put*, *pipe* and *get* respectively correspond to transferring the image data from disk to the local memory on the CHS2x4 board, piping the data through the FPGA circuitry, and getting the results from the board. Even with the slow software-controlled FPGA execution, a speedup of over 20 times can be obtained if data transfer overhead to and from disk is not included (software time/*pipe* time), while a speedup of a factor of two is observed if we include the data transfer overhead. Notice that the total time for the hardware-assisted version is not equal to the sum of the time required for *put*, *pipe* and *get*; the difference (given by *others*) accounts for the time spent on dividing the image into sub-images and combining the results, and on routines for timing the program. Since this length of time is around 5% of the total processing time, our partitioning strategy appears to incur a modest overhead for this system.

| Image size (pixels) | Software | Hardware and software | |
|---|---|---|---|
| 25600 | | *put* | 0.45 |
| (160x160) | | *pipe* | 0.16 |
| | | *get* | 0.55 |
| | | *others* | 0.06 |
| Total time (seconds) | 2.19 | | 1.22 |
| 40960 | | *put* | 0.71 |
| (160x256) | | *pipe* | 0.21 |
| | | *get* | 0.89 |
| | | *others* | 0.11 |
| Total time (seconds) | 4.28 | | 2.19 |
| 50176 | | *put* | 0.88 |
| (224x224) | | *pipe* | 0.28 |
| | | *get* | 1.04 |
| | | *others* | 0.10 |
| Total time (seconds) | 4.39 | | 2.30 |
| 84956 | | *put* | 1.55 |
| (268x317) | | *pipe* | 0.39 |
| | | *get* | 1.80 |
| | | *others* | 0.21 |
| Total time (seconds) | 7.09 | | 3.95 |

**Table 1** Comparing performance of Sobel edge detection in software and in hardware/software.

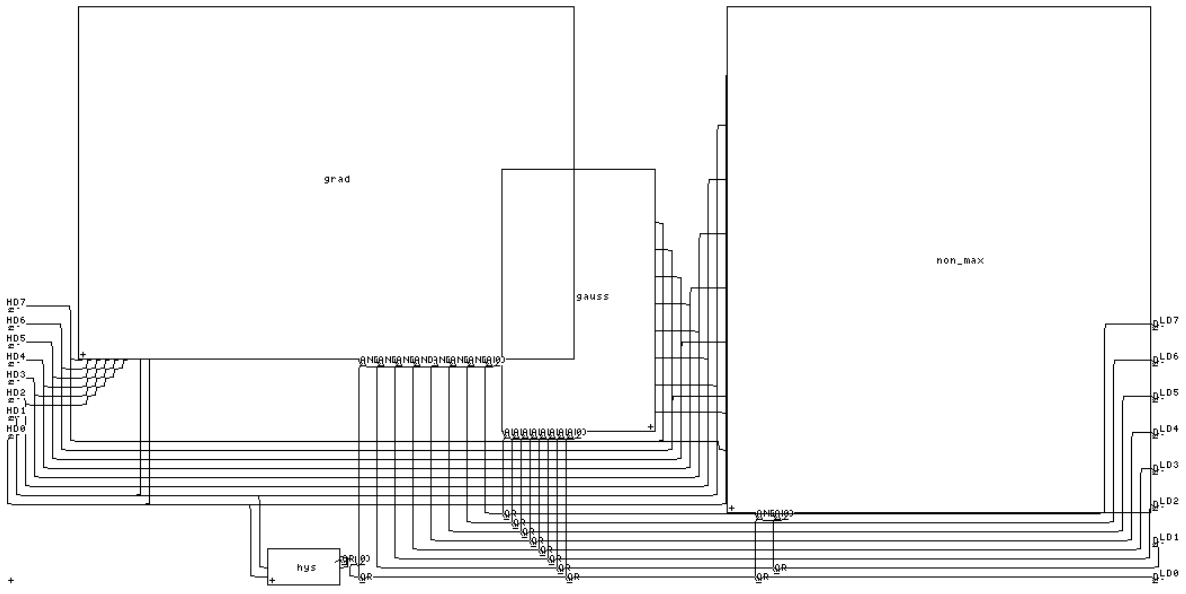

**Figure 3** Hardware design for the Canny edge detector.

## 8 The Canny Edge Detector

The Canny algorithm [3] is a more elaborate edge detector than the Sobel algorithm. It consists of four stages: two-dimensional Gaussian smoothing for removing noise in the image, gradient calculations for extracting the potential edge points, suppression of non-maximum edge points for generating local maxima, and thresholding and hysteresis of the local maxima for recovering broken edges.

Our hardware design for the Canny edge detector is shown in Figure 3. It consists of four blocks, `gauss`, `grad`, `non_max` and `hys`, which are invoked one at a time by dynamically reconfiguring the control logic, such that their inputs are taken from the input port on the left and their outputs are multiplexed by or-gates (labelled "OR" in the diagram) to the output port on the right. The overlapped region reflects an optimisation in the use of FPGA cells; there are no connections between circuits belonging to different blocks. These blocks were developed by hand, although there are now various compilers that can be used to automate their production [8].

The block labelled `gauss` contains adders and shifters for implementing a Gaussian filter using the mask

$$[[1,2,1],[2,4,2],[1,2,1]].$$

The `grad` block computes the image gradient by calculating the difference in magnitude between a given pixel and its four neighbours; circuits for arithmetic and multiplexing are included. Non-maximum suppression and thresholding are performed by the `non_max` block, which is an array of registers, comparators, multiplexers and a multiplier. The `hys` block is responsible for hysteresis and is constructed from several gates and registers. Example input and output images are given in Figure 4.

The final stage of Canny's algorithm, hysteresis, complicates the merging of results from sub-images. For reasons that we shall not go into here, the pixels near the edges of the image and neighbouring sub-images must be combined using the logical "or" operation. This is an example where merging is performed by a binary operator other than $+\!\!+$.

Our implementation involves several arrays, each of length equal to the maximum size of sub-images, to store intermediate results. To reduce memory usage even further, we only read in part of the image at any time instead of the whole image. Partial edge results are written to a file so that the array can be reused to store results of the next sub-image. This method permits an image of arbitrary size to be processed, although the program runs slower as there are more accesses to disk. Figure 5 compares the processing time for a software Canny program against that for a hardware-assisted version for images of various sizes. The processing time by the software version is non-linear in the number of pixels of the images, since it increases with the number of edges in the image. In contrast, the processing time by the hardware-assisted version increases linearly with the number of pixels and is independent of the number of edges. This explains why the software version sometimes outperforms the hardware-assisted version: it happens when there are relatively few edges in the image. In general, the hardware-assisted version is around 39% faster than the software version.

Notice that around 88% of the measured time is devoted to communication with the PC; the corresponding times for the Sobel edge detector are given by the *put* and *get* figures in Table 1. If this overhead is not included, then the hardware-assisted design is approximately 13 times faster than the software version. Furthermore, if the input-output bottleneck can be eliminated so that the only speed limitation is the critical path delay, we estimate that a speedup of about 300 times can be achieved. The critical path of the hardware can be reduced by improving the layout and by adding latches along the critical path – at the expense of increasing power consumption and latency. This will be necessary if the edge detector is to operate at video rate.

## 9 Summary

A codesign strategy has been presented for developing flexible and efficient implementations for systems containing both hardware and software components. A framework based on functional programming has been used in describing and verifying the hardware-software partitioning strategy. This approach facilitates the production of parametrised realisations with a range of cost/performance trade-offs. Our experiments indicate that for a number of applications, the overhead associated with the partitioning strategy is acceptable.

Further work includes studying the use of other formalisms for multidimensional arrays (such as [10]) and other data structures, which may simplify program description and verification. The refinement from such high-level descriptions to a hardware notation – for instance Ruby (see [8],[9]) – would be investigated.

We are also exploring the theory and practice of generating the hardware, the software and their interface from high-level descriptions. Concurrency and synchronisation can be captured effectively in a framework based on CSP [4], and our HARP system [11], which contains a Xilinx FPGA, a transputer and other components, would be used in experimenting with different variations of our partitioning method.

**Acknowledgements**

The hardware Sobel edge detector was implemented by Steve Chiu. Thanks to Mark Jones, John O'Leary, Mike Spivey and Alan Wenban for comments and suggestions. The support of Oxford Parallel Applications Centre, Esprit OMI/HORN project, Scottish Enterprise and Xilinx Development Corporation is gratefully acknowledged.

## References

[1] Algotronix Ltd, *CHS2x4 Custom Computer User Manual*, 1992.

[2] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice-Hall International, 1988.

[3] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No.6, pp. 679–698, November 1986.

[4] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985.

[5] M.P. Jones, *An Introduction to Gofer*, Oxford University Programming Research Group, 1991.

[6] T. Kean and J. Gray, "Configurable hardware: two case studies of micro-grain computation," in *Systolic Array Processors*, J.V. McCanny, J. McWhirter and E.E. Swartzlander Jr. (eds.), Prentice Hall, 1989, pp. 310–319.

[7] S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[8] W. Luk and I. Page, "Parameterising designs for FPGAs," in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 284–295.

[9] W. Luk, V. Lok and I. Page, "Hardware acceleration of divide-and-conquer paradigms: a case study," in *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1993, pp. 192–201.

[10] R. Miller, *A Constructive Theory of Multidimensional Arrays*, Oxford University Programming Research Group, 1993.

[11] I. Page and W. Luk, "Compiling occam into FPGAs," in *FPGAs*, W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1991, pp. 271–283.

[12] A. Wenban, J. O'Leary and G.M. Brown, "Codesign of communication protocols," *IEEE Computer*, Vol. 26, No. 12, pp. 46–52, December 1993.

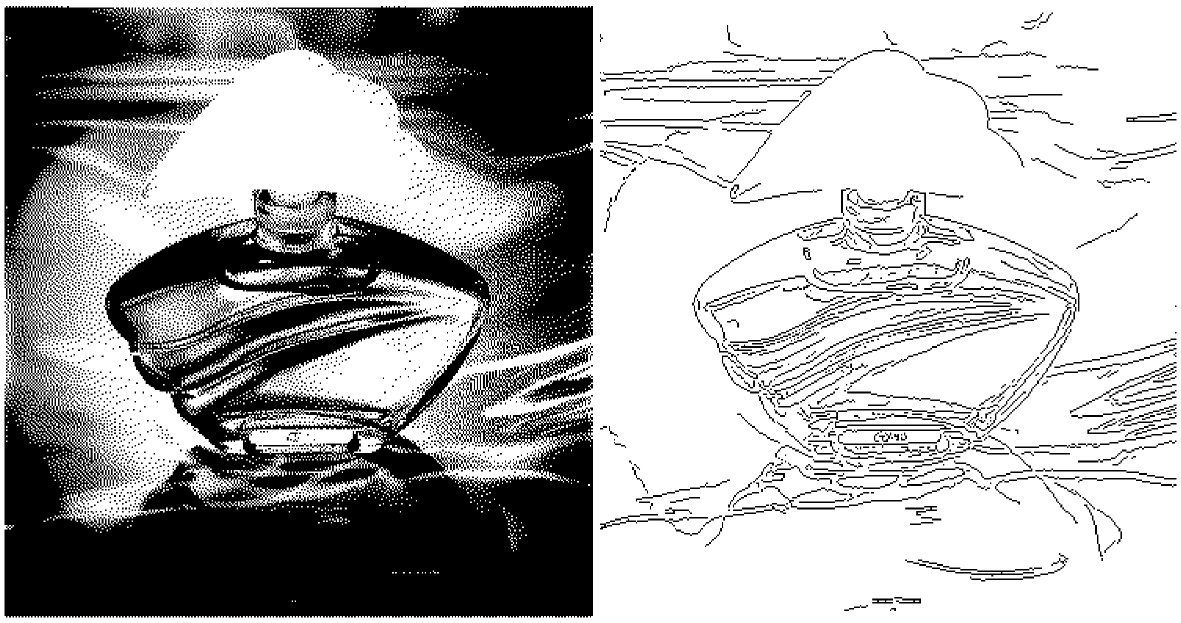

**Figure 4** The image “Perfume” and the result of applying Canny’s algorithm.

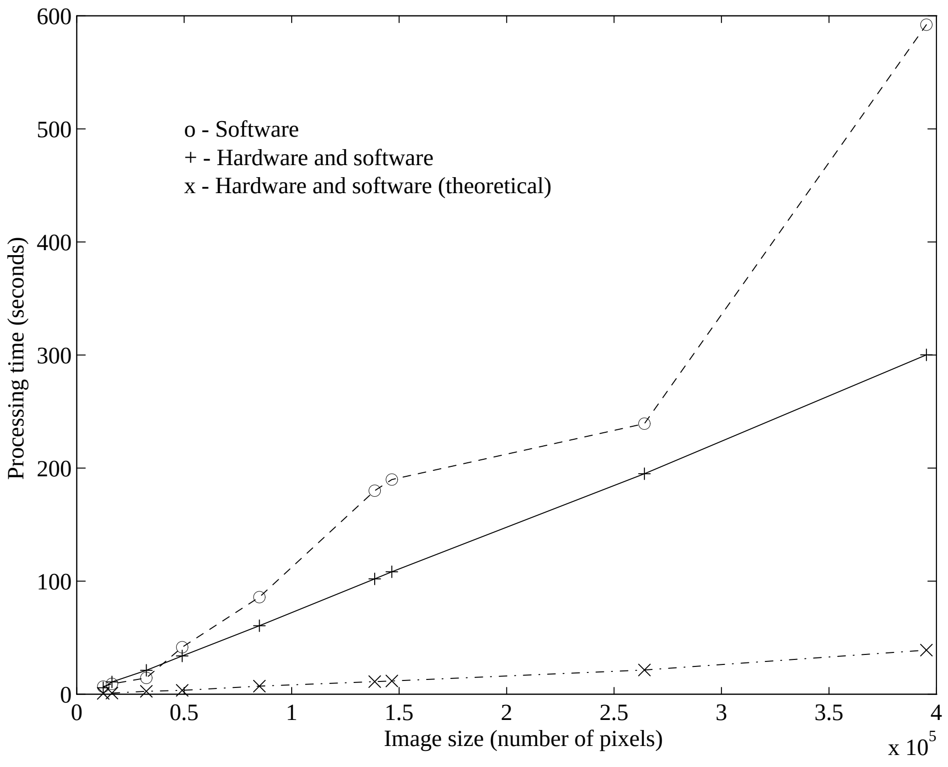

**Figure 5** Comparing performance of our implementations of Canny’s edge detector.