Hardware Acceleration of Divide-and-Conquer Paradigms: a Case Study

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

Abstract

We describe a method for speeding up divide-and-conquer algorithms with a hardware coprocessor, using sorting as an example. The method employs a conventional processor for the "divide" and "merge" phases, while the "conquer" phase is handled by a purpose-built coprocessor. It is shown how transformation techniques from the Ruby language can be adopted in developing a family of systolic sorters, and how one of the resulting designs is prototyped in eight FPGAs on a PC coprocessor board known as CHS2x4 from Algotronix. The execution of the hardware unit is embedded in a sorting program, with the PC host merging the sorted sequences from the hardware sorter. The performance of this implementation is compared against various sorting algorithms on a number of PC systems.

1 Introduction

It has long been recognised that the performance of a conventional processor can be speeded up many times if computationally-intensive operations are delegated to purpose-built hardware. Such hardware accelerators may even be put on the same chip as the processor itself – the inclusion of a floating-point unit in the Inmos T805 transputer and the Intel 486 microprocessor are well-known examples. There is, however, a limit on the number of specialised units that can reasonably be attached to the main processor. RAM-based Field-Programmable Gate Array (FPGA) technology offers an attractive solution: an array of FPGAs, closely coupled to a host microprocessor, can be reconfigured rapidly as a special-purpose coprocessor for many applications.

What factors should one consider when implementing algorithms on FPGA-based coprocessors? There are, of course, the features of the FPGA itself – the number and function of cells and their interconnection structure – as well as the size and shape of the computation array if more than one FPGA is employed. Another important factor is the way that data are passed through the hardware: the speed, location and size of the input and output ports. The availability of appropriate design methods and tools must also not be overlooked. Our objective is to study a strategy for mapping divide-and-conquer algorithms onto an FPGA-based coprocessor that takes the above factors into account.

A number of hardware accelerators based on FPGAs have been reported recently. Some are built mainly for a particular application such as gene sequence analysis [4], while others are designed to be general-purpose (see [2], [5]). Our work involves a commercially available system from Algotronix known as CHS2x4 [1]. It 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 [7] 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. Figure 1 shows a CAL cell with its function unit configured as an and-gate taking inputs from the north and west and output to the east; the spare ports on the north and east are used for routing.

In the following sections we review the partitioning strategy and the architectural development route for implementing a hardware accelerator for sorting on the CHS2x4.

2 Partitioning strategy

The first problem is to decide how the algorithm can be partitioned in two: one that will be implemented in

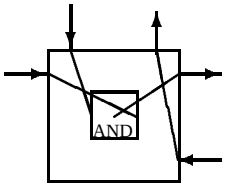


Figure 1 An example CAL cell.

software running on the host processor, and the other on the coprocessor board. It is clear that the host processor is suitable for executing relatively complex, data-dependent and irregular operations, while the hardware unit – because of the array structure of CAL – can best be exploited in a systolic mode.

A simple but promising approach for partitioning divide-and-conquer algorithms is to employ the host processor for the "divide" and "merge" phases, and the coprocessor for the "conquer" phase. This approach is particularly effective if the "conquer" phase is computationally demanding. An obvious solution is for the host processor to divide the data into fixed-size blocks that can be handled by the dedicated hardware; possible applications include signal and image processing, where the amount of data is often too large for the coprocessor to handle at once.

This paper illustrates the above partitioning strategy using a simple example: sorting. Devising the task for the host processor was straightforward; it would divide the data into fixed-size blocks for the hardware sorter, and would merge the sorted sequences from the hardware sorter. The selection of appropriate sorting algorithms for the CHS2x4 board, however, needs careful consideration.

Algorithms with good asymptotic behaviour but requiring non-uniform data movements or a bulky control unit, such as quicksort, mergesort or heapsort, cannot be efficiently mapped onto the CHS2x4 board. Regular networks like odd-even sort and bitonic sort are possible candidates, but they work best if data can be input and output in parallel, and their connection structures are more complicated than the orthogonal grid that CAL offers. In the end we choose to implement a systolic insertion sorter, which reads a given sequence in a bit-parallel word-serial format, and produces the sorted sequence in the same manner. This method conforms to the hardware constraints of the CHS2x4 board, provided that the data to be sorted do not exceed the width of the input and output ports – in this case eight bits.

While the "divide" and "merge" phases for our sorter implementation involve simply partitioning and merging of sequences of numbers, this may not be the case for some applications such as image processing. For instance, in convolution-based edge detection it is necessary to replicate part of the image to overcome boundary effects in processing partitioned images. A method has been developed to ensure that partitioning will not alter the result of the computation; the details will be covered in a forthcoming publication.

3 Architectural development

We now describe how transformation techniques from the Ruby language can be adopted in developing a family of systolic sorters. These techniques can be regarded as a systematic generalisation and formalisation of design experience; they also provide a basis for computer-based tools (see [12], [13]) for design development and implementation. A fuller description of Ruby can be found elsewhere (for instance [6], [9], [10]), and we shall only outline the five steps in developing a generic description of an architecture based on insertion sort.

First of all, note that insertion sort can be described by the function $insort\ a$ which takes a sorted sequence and inserts the element a at the appropriate place with respect to the ordering relation:

$$\begin{aligned} insort\ a\ \langle\,\rangle &= \langle a\rangle, \\ insort\ a\ (\langle x\rangle\hat{}xs) &= \mathsf{if}\ a \le x \\ &\quad\ \mathsf{then}\ \langle a, x\rangle\hat{}xs \\ &\quad\ \mathsf{else}\ \langle x\rangle\hat{}insort\ a\ xs \\ &\quad\ \mathsf{fi}. \end{aligned}$$

So $insort\ 3\ \langle 0,1,2,5\rangle = \langle 0,1,2,3,5\rangle$. Here "^" denotes an operation for appending two sequences, for instance $\langle a,b,c\rangle\hat{}\langle d,e\rangle = \langle a,b,c,d,e\rangle$. Another function on sequences that we shall use is apr (append right), which appends an element to the end of a sequence: $apr\ \langle\langle a,b,c,d\rangle, e\rangle = \langle a,b,c,d,e\rangle$.

Next, we shall recast the above recursion equations in an algebraic form so that optimising transformations can be applied. This is achieved by implementing $insort$ as a row of leaf cells. In Ruby leaf cells are described by binary relations, so that a doubling operation can be described by $x\ double\ 2{\times}x$, where x is the domain and $2{\times}x$ is the range of the relation $double$. Note that $double$ can also be used to describe a divide-by-2 operation, if the range is regarded as the input.

Before introducing the operator that corresponds to a row of cells, let us consider how two adjacent circuits can be put together in series and in parallel. Two circuits Q and R in series is denoted by $Q;R$, a composite circuit with Q and R sharing a hidden compatible interface s (Figure 2a):

$$x\ (Q;R)\ y \quad \Leftrightarrow \quad \exists s.\ (x\ Q\ s)\ \&\ (s\ R\ y),$$

so $x\ (double; double)\ 4 \times x$. The ";" operator is known as relational composition, and can easily be shown to be associative. Parallel composition of two components Q and R, denoted by $[Q, R]$, represents the combination with no connection between Q and R (Figure 2b),

$$\langle x, y\rangle\ [Q, R]\ \langle u, v\rangle \quad \Leftrightarrow \quad (x\ Q\ u)\ \&\ (y\ R\ v),$$

so $\langle x, y\rangle\ [double, (double; double)]\ \langle 2 \times x, 4 \times y\rangle$. Given that ι is the identity relation, we have the abbreviations $\mathsf{fst}\ R = [R, \iota]$, and $\mathsf{snd}\ R = [\iota, R]$.

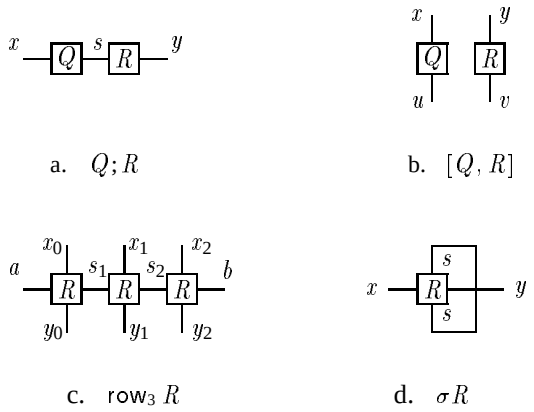


Figure 2 Pictures of some Ruby operators.

A cascade of R's is given by R^n, defined inductively by the equations $R^1 = R$ and $R^{n+1} = R^n\ ;\ R$. A row of components with vertical and horizontal connections (Figure 2c) can be described by the row operator (note that $\#x$ denotes the number of elements in sequence x),

$$\begin{array}{l}\text{if } \#x = \#y = N\\ \text{then } \langle a, x\rangle\ (\mathsf{row}_N\ R)\ \langle y, b\rangle \quad \Leftrightarrow\\ \qquad \exists s.\ (s_0 = a)\ \&\ (s_N = b)\\ \qquad\quad \&\ \forall i : 0 \le i < N.\ \langle s_i, x_i\rangle\ R\ \langle y_i, s_{i+1}\rangle.\end{array}$$

Given operators such as relational composition and row that encapsulate common patterns of computations, one can show that

$$insort\ a\ xs \quad = \quad (\mathsf{row}_n\ scell\ ;\ apr)\ \langle a, xs\rangle$$

where $\#xs = n$ and $scell\ \langle a, x\rangle = \mathsf{if}\ \ a \le x\ \mathsf{then}\ \langle a, x\rangle\ \mathsf{else}\ \langle x, a\rangle\ \mathsf{fi}$.

We now come to the third step in developing the systolic sorters. The row of $scell$s defines its state-transition logic, and to describe the complete circuit we need a way of representing latches and feedback loops, and sequential circuits in general. This is achieved in Ruby by relations that handle streams, or time sequences, such that the stream version of $double$ becomes $x\ \ double\ \ y \Leftrightarrow (\forall t.\ 2 \times x_t = y_t)$. It can be shown that the algebraic properties of Ruby are preserved by lifting relations to work on streams. A latch is modelled by a relation $\mathcal{D}$ whose range stream is one time unit behind the domain stream, $x\ \mathcal{D}\ y \Leftrightarrow (\forall t.\ x_{t-1} = y_t)$. For any R such that $\mathcal{D}; R = R; \mathcal{D}$ (which is the case, for instance, when R is combinational), the distributive law $R^n; \mathcal{D}^n = (R; \mathcal{D})^n$ holds. Like ι, $\mathcal{D}$ can be used for any type of signals, so $\mathcal{D}; [R, R] = [\mathcal{D}, \mathcal{D}]; [R, R]$ and

$$(\mathsf{row}_n\ R)\ ;\ \mathsf{fst}\ \mathcal{D} \quad = \quad \mathsf{row}_n\ (R\ ;\ \mathsf{fst}\ \mathcal{D}). \qquad (1)$$

To describe a circuit with feedback, we use the operator σ, given by $x\ \ \sigma R\ \ y \Leftrightarrow \exists s.\ \langle x, s\rangle\ \ R\ \ \langle s, y\rangle$ (Figure 2d). Since a state machine with state-transition logic Q can be represented by $\sigma\ (Q; \mathsf{fst}\ \mathcal{D})$, a serial sorter can be obtained in the same way by adding latches and feedback paths to the row of combinational sorters $\mathsf{row}_n\ scell$ to give

$$\begin{array}{rcl} sort1 & = & \sigma((\mathsf{row}_n\ scell)\ ;\ \mathsf{fst}\ \mathcal{D})\\ & = & \sigma(\mathsf{row}_n\ (scell\ ;\ \mathsf{fst}\ \mathcal{D}))\end{array}$$

by Equation 1. Note that the correct operation of $sort1$ requires the feedback latches to be initialised with the greatest element given by the ordering relation.

Right now $sort1$ is expressed as a single state machine with a single bank of feedback latches. The fourth step in the development process is to decompose this state machine into a cascade of simple state machines, which can then be pipelined so that the clock speed is independent of the number of processors. We apply the theorem

$$\sigma\ (\mathsf{row}_n\ R) \quad = \quad (\sigma\ R)^n \qquad (2)$$

for state machine decomposition (Figure 3), whose correctness is proved in [10], so that $sort1$ becomes $(\sigma(scell\ ;\ \mathsf{fst}\ \mathcal{D}))^n$.

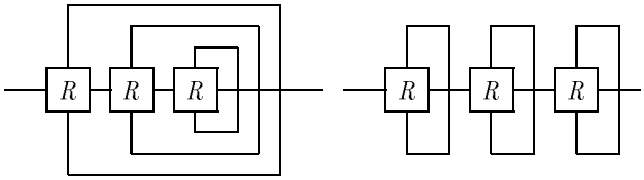


Figure 3 Decomposing a large state machine into a cascade of small state machines.

Given that $n = km$, the final step is to pipeline $sort1$ by the theorem $R^{km}; \mathcal{D}^m = (R^k; \mathcal{D})^m$ (provided that $\mathcal{D}; R =$

$R;\mathcal{D}$) to give

$$sort2 \quad = \quad ((\sigma(scell\ ;\ \mathsf{fst}\,\mathcal{D}))^k\ ;\ \mathcal{D})^m,$$

since composing one or more $\mathcal{D}$'s with $sort1$ only increases its latency and does not change its behaviour. Notice that the parameter k controls the degree of pipelining: with $k = 1$ and $m = n$, $sort2$ becomes a fully-pipelined design, otherwise signal rippling through k $scell$s will occur. Moreover, $sort2$ has a latency of $m + n = n(k + 1)/k$ cycles and requires $2m + n = n(k + 2)/k$ latches; hence a smaller k results in a faster circuit, but the latency and the number of latches in the design will increase.

This completes the development of a word-level description of a family of sorters. In the next section, we shall present the implementation of a fully-pipelined version of $sort2$, which consists of a cascade of cells each containing an $scell$, a latched feedback path, an input A and a latched output Max (Figure 4, with latches represented by heavy dots).

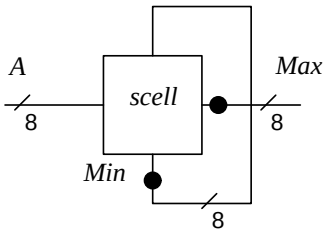


Figure 4 Repeated unit of the systolic sorter.

4 Bit-level design and implementation

Figure 5 contains the block diagram of a bit-level cell which can be used to implement an $scell$ with feedback as shown in Figure 4. Notice that an $init$ wire is introduced for presetting the latch to a desired value. The design can be divided into three smaller units: a comparator, a double multiplexer and an initialisation unit. The correctness of this refinement step, which relates a word-level description and its bit-level implementation, is outlined in [11] for the comparator. Although the CAL circuits shown later were developed by hand, there are now various compilers that can be used to automate their production [13].

The ith-bit comparator takes two inputs A_i and B_i, the ith-bit of the two numbers to be compared, and returns the result C_{i+1}, where

$$C_{i+1} = A_i\overline{B_i} + C_iA_i + C_i\overline{B_i}.$$

The corresponding truth table is as follows:

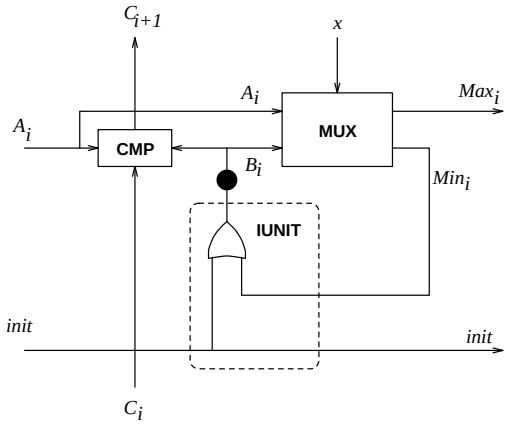


Figure 5 Bit-level sorter cell.

A_i	B_i	C_{i+1}
1	0	1
0	1	0
0	0	C_i
1	1	C_i

If $A_i > B_i$, C_{i+1} will be set to logic one; if $A_i < B_i$, C_{i+1} will be set to logic zero. For the case $A_i = B_i$, the output C_{i+1} will be the same as C_i, the result from the comparison of the previous bit.

As we form a column of m comparators as shown in Figure 6 with C_0 hardwired to logic one, the output at C_m, the most-significant bit, will give the final result of the comparison: $C_m = 1$ means $A \geq B$ and $C_m = 0$ means $A < B$. This final result is then broadcast to every multiplexer unit as the signal x in Figure 5. Figure 7 shows the corresponding CAL design.

The double multiplexer steers the two input bits A_i and B_i to the output Max_i and Min_i according to the input signal x. When $x = 1$ (namely $A \geq B$), A_i will be connected to Max_i and B_i connected to Min_i. The connection will be the other way round for the case $x = 0$. Figure 8 and Figure 9 contain the truth table and the logic equation along with the CAL design.

The initialisation unit consists of an or-gate as shown in Figure 5. Its purpose is to initialise the latch on the feedback path to the maximum value of the elements to be sorted, by employing the $init$ signal to preset B_0 to B_{m-1} to logic one. to close the feedback path from the Min_i output of the double multiplexer to its B_i input.

Putting together the comparator, the double multiplexer and the initialisation unit as shown in Figure 5, we arrive at a bit-level basic cell which takes up 6 by 6 CAL cells. Since all the bit-level cells within the same word share the

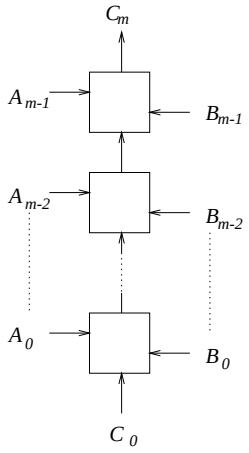


Figure 6 Bit-level comparator structure.

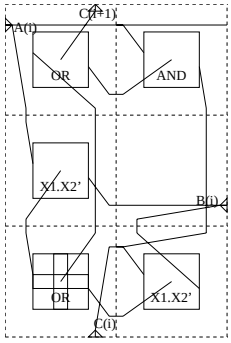
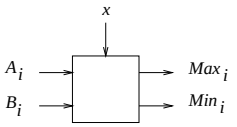


Figure 7 CAL design of a comparator cell.



$x = 0 \rightarrow A < B$

$x = 1 \rightarrow A \geq B$

x	Max_i	Min_i
0	B_i	A_i
1	A_i	B_i

Figure 8 The double multiplexer design.

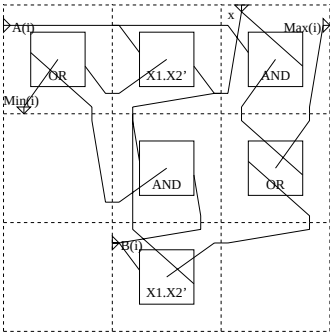


Figure 9 The CAL design of the double multiplexer.

same $init$ signal, we only need one pair of latches on the $init$ line for the entire bit-level structure. The final design in Figure 10 occupies 6 by 3 CAL cells with only two dedicated to routing. A design without the latches at the Max output will probably require the same number of CAL cells, and it would have a lower latency but a longer critical path than our fully-pipelined systolic sorter.

The complete sorter design consists of a pipeline of latched $scell$s, based on the repeated unit shown in Figure 4. The more repeated units we have, the longer the pipeline and the longer the sequence the design can sort. So the final task is to fit a maximum number of repeated units onto the CHS2x4 board with the input and output properly wired to the I/O bus, keeping the length of all the wires to a minimum. Figure 10 shows the board layout with 38 repeated units for a design which can sort sequences containing up to 39 elements. Note that the cells on the upper-half of the design are a reflected version of the one shown at the top of Figure 10.

This design has been further optimised by adopting a scheme, similar to the use of tag bits in the Splash 2 system [3], which enables the continuous processing of a sequence of elements of arbitrary length. The input data pass through the hardware sorter once and emerge as a stream of sorted blocks, each containing 39 elements.

5 Host software and performance evaluation

The software to drive the CHS2x4 board consists of a series of calls to the interface library supplied by Algotronix. It first resets the board to clear all previous programming information for the cells as well as for the internal registers. It then programs the board with the configuration file associated with the design. Next, the elements to be sorted are stored in the on-board memory. After initialisation, the memory counter is set to point to the address of the first

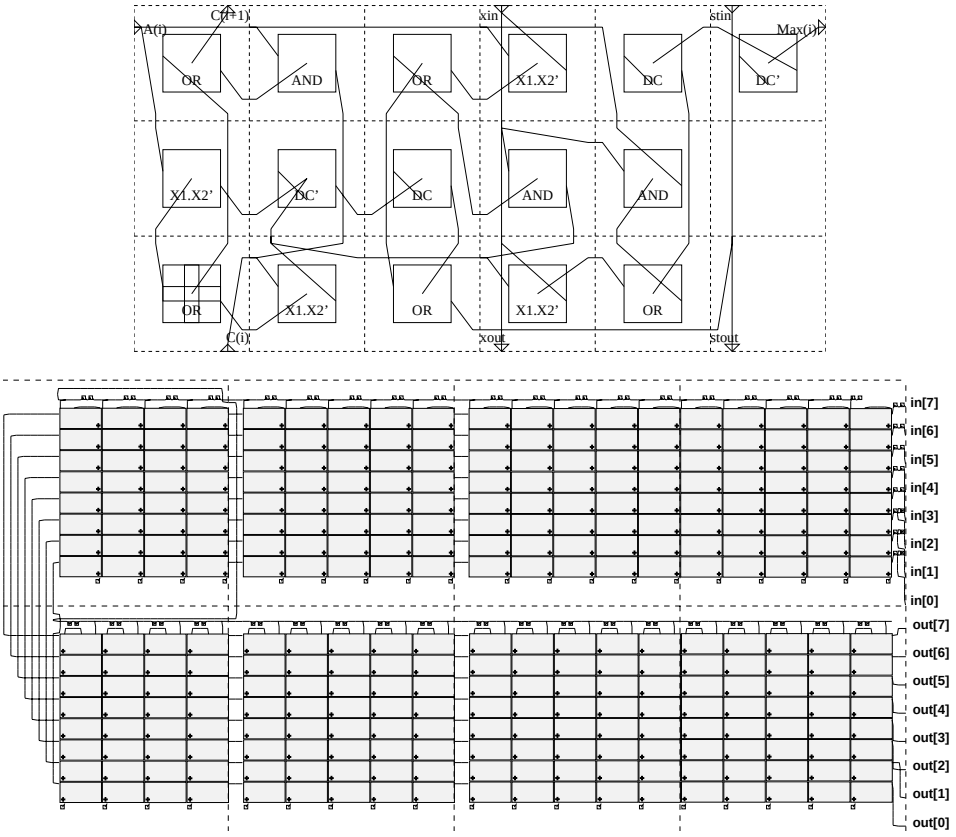


Figure 10 Top, optimised unit cell design; bottom, the board layout. Every small shaded box in the diagram represents a unit cell. The dash lines indicate the chip boundary.

input data. The computation loop can now begin, with data circulating between the FPGAs and the on-board memory. When completed, the output data can be collected from the on-board memory. The host processor then computes the final answer by combining the sorted sequences from the coprocessor. Figure 11 summarises the operation.

This implementation, `hsort`, is evaluated and compared with other algorithms that have been implemented in software for sorting 65400 7-bit random numbers. (One of the bits in the byte-wide channel between the FPGAs and the on-board memory is used as a control signal – hence 7-bit data – but it would be straightforward to generate this control signal using some of the FPGAs). First, Table 1 compares the time required for sorting 1677 sequences each containing 39 7-bit numbers, using the insertion sorter in `hsort` and an insertion sort program running on the host. The timing results are obtained by using the `ftime` command in Microsoft C library, which is assumed to be accurate to within 0.1 second. The variation in speed of the hardware sorter in different hosts is too small to be measured accurately; the speed of transferring data between the on-board memory and the FPGAs is estimated to be around 0.65 megabytes per second. The gain in speed over the software sorter ranges from 12 times on a 486-based machine, to 233 times on a 286-based machine. These timing results, however, do not include the time required to merge the sorted sequences nor, in case of the hardware sorter, the time required for transferring data between the PC and the memory on the CHS2x4 board.

Next, Table 2 summarises the proportion of time for hardware sorting and for merging the sorted sequences. Note that the timing results do not include the time for configuring the CAL chips on the CHS2x4 board. The table confirms that the time for merging dominates the sorting process, and this time decreases with a faster machine. In other words, faster machines spend a larger proportion of their time on sorting in hardware, which is consistent with the results shown in Table 1.

Table 3 consists of an analysis of `chsort`, which is the same as `hsort` except that the time for configuring

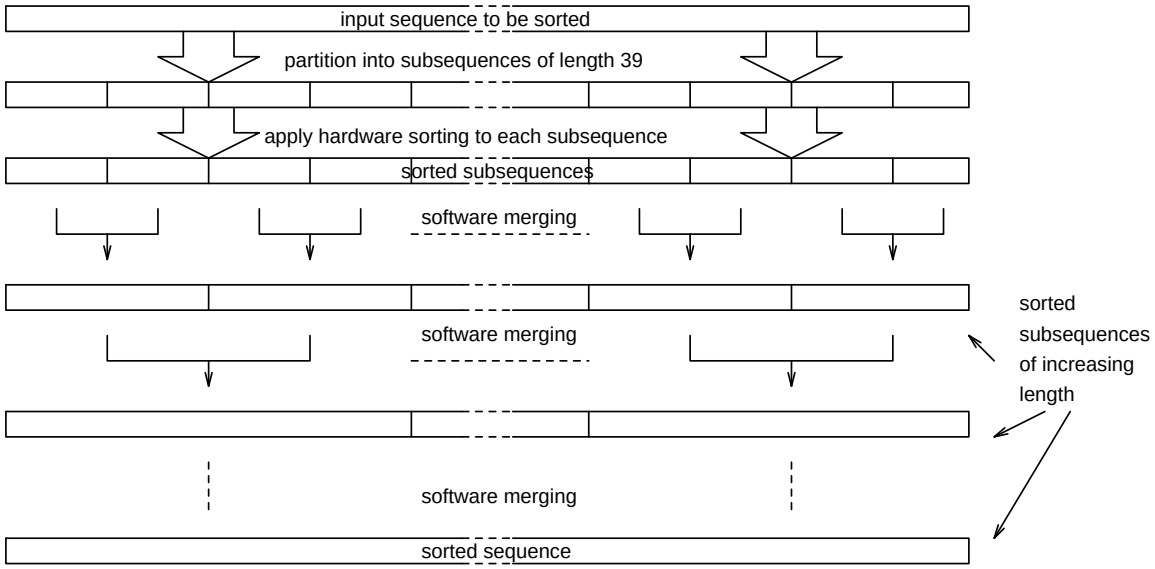


Figure 11 Implementing "divide" and "merge" in software and "conquer" in hardware.

Table 1 A comparison of the time required, in seconds, for sorting 1677 sequences each containing 39 7-bit numbers, using insertion sort in software and in hardware.

host	software sort	hardware sort
286 turbo off	23.3	0.1
286 turbo on	11.1	0.1
386 turbo off	15.4	0.1
386 turbo on	5.9	0.1
486 turbo off	3.6	0.1
486 turbo on	1.2	0.1

the eight FPGAs is included. The table shows that the faster machines spend a larger proportion of their time on configuring the FPGAs.

Finally, Table 4 compares the performance of `hsort` and `chsort` with `qsort`, a quicksort procedure [15], `msort`, a mergesort program, `qsortlib`, the "qsort" routine from Microsoft C library, and `dsort`, a linear sorting algorithm based on distributed counting [16]. `qsort` demonstrates its superiority by beating `hsort` for the slower machines, although `hsort` wins by 20% for the faster hosts. `msort` employs a non-recursive merging procedure similar to the one used in `hsort` – this shows that the hardware sorter speeds up the sorting process by up to 30%. Since in `hsort` the software merging requires $\lceil log_2 (65400/39) \rceil = 11$ passes over the sorted subsequences, it is an order of magnitude slower than the linear time `dsort` algorithm; however, for a sequence of n m-bit numbers, `dsort` requires an array of 2^m variables with $log\ n$-bits each, so its use is restricted to sorting long sequences of relatively small numbers. In contrast, `hsort` would be more effective for sorting relatively short sequences of large numbers. The byte-wide interface between the FPGAs and the on-board memory of the CHS2x4 system, however, complicates the implementation of sorters for large numbers.

To summarise, from these tables it is apparent that there are three major performance bottlenecks in the current `hsort` implementation, excluding the time required for configuring the FPGAs. Two of them are related to the interface between the hardware coprocessor and the host processor: the time to load and unload the on-board memory, and the slow software-controlled interface between the FPGA array and the on-board memory. The third, and the most severe bottleneck, is the time required for merging the sorted sequences in software. Some suggestions for reducing the effects of these bottlenecks will be described in the next section.

Table 2 An analysis of hardware-accelerated sorting of 65400 7-bit numbers, showing the relative amount of time spent on sorting with the FPGAs, on data transfer between the on-board memory and the PC, and on merging the sorted sequences by the host.

host	`hsort` (seconds)	hardware sort	data transfer	software merge
286 turbo off	60.3	0.2%	7.6%	92.2%
286 turbo on	28.7	0.2%	7.6%	92.2%
386 turbo off	24.2	0.3%	10%	89.7%
386 turbo on	9.5	0.7%	10%	89.3%
486 turbo off	7.7	1.5%	16.4%	82.1%
486 turbo on	2.1	3.2%	15.9%	80.9%

Table 3 Effect of configuration time on hardware accelerated sorting of 65400 7-bit numbers.

host	`chsort` (seconds)	hardware sort	data transfer	software merge	FPGA configuration
286 turbo off	77.0	0.1%	5.9%	72.1%	21.8%
286 turbo on	37.1	0.2%	5.9%	71.3%	22.7%
386 turbo off	36.7	0.2%	7.6%	68.0%	24.2%
386 turbo on	14.6	0.5%	7.5%	67.2%	24.7%
486 turbo off	11.4	0.8%	7.7%	64.4%	27.1%
486 turbo on	3.6	2.3%	11.2%	57.3%	29.2%

Table 4 A comparison of the time required – in seconds – for various implementations to sort 65400 7-bit numbers; the quantities in bracket are the slowdown factors relative to `hsort`.

host	`hsort`	`chsort`	`qsort`	`msort`	`qsortlib`	`dsort`
286 turbo off	60.3 (1)	77.0 (1.3)	53.7 (0.9)	80.2 (1.3)	545.6 (9.0)	3.3 (0.05)
286 turbo on	28.7 (1)	37.1 (1.3)	24.4 (0.9)	38.7 (1.3)	262.6 (9.1)	1.5 (0.05)
386 turbo off	24.2 (1)	36.7 (1.5)	30.0 (1.2)	28.5 (1.2)	378.4 (15.6)	1.8 (0.07)
386 turbo on	9.5 (1)	14.6 (1.5)	11.5 (1.2)	10.9 (1.1)	161.1 (17.0)	0.7 (0.07)
486 turbo off	7.7 (1)	11.4 (1.5)	9.5 (1.2)	9.2 (1.2)	125.0 (16.2)	0.5 (0.06)
486 turbo on	2.1 (1)	3.6 (1.7)	2.6 (1.2)	2.2 (1.0)	43.3 (20.6)	0.2 (0.10)

`hsort`: hardware-accelerated sorting with merging performed in software;
`chsort`: same as `hsort` but includes the time for configuring the FPGAs;
`qsort`: quicksort;
`msort`: mergesort;
`qsortlib`: "qsort" routine in Microsoft C library;
`dsort`: sorting program based on distributed counting.

6 Concluding remarks

We have discussed a method of partitioning divide-and-conquer algorithms and illustrated how it can be implemented on a commercially available FPGA-based hardware platform. The deployment of this method for other applications such as image processing is currently being investigated. Our experience confirms that it is reasonably straightforward to prototype hardware designs using FPGA technology: part of the detailed design and evaluation work of the pipelined sorter was carried out as part of a 200-hour undergraduate project.

While experimental results demonstrate the viability of our approach, a hardware accelerator can only remain competitive if it performs at least an order of magnitude faster than the equivalent software rival. From analysing the critical paths in our design using data from the manufacturer, we estimate that the hardware sorter is capable of running at a maximum speed of 2.5 MHz; further pipelining at bit-level can deliver another six-fold improvement. One way of exploiting this potential is to speed up the interface between the FPGA array and the memory on the coprocessor board, for instance by clocking the FPGAs at a higher speed with interrupt-driven data transfer between the coprocessor and the host processor. This enhancement will allow the host processor to run concurrently with the coprocessor; however, some hardware modifications to the CHS2x4 board will be necessary. Other hardware alterations that should improve performance include widening the width of input and output ports, increasing the amount of the on-board memory and allowing it to be addressed directly from the host processor.

A more effective implementation of the "divide by software and conquer by hardware" methodology for sorting long sequences will involve implementing the merging process in hardware, because that is what dominates the processing time and what needs to be conquered. Since the merging process is largely sequential and requires non-linear access to the on-board memory, it will be inefficient to carry out in the current version of the CHS2x4 system. It should be possible to use our occam compiler [14] to speed up the merging process on a HARP1 board under development at Oxford, which contains a Xilinx FPGA tightly-coupled to a transputer. For other applications, software may still be adequate for data partitioning and merging: some preliminary results suggest that partitioning and merging data in software for an edge detector, for instance, occupies less than 10% of the total processing time.

It should be noted that hardware-assisted processing is probably not shown to full advantage in this case study, because microprocessors are already fairly good at algorithms such as sorting which are memory-bandwidth limited; so we can anticipate some larger speedups with algorithms exhibiting a larger computation/communication ratio.

The use of Ruby for developing and implementing hardware designs has been encouraging, and will become increasingly important in the future. In particular, a Ruby description can be parametrised in various ways for generating designs of different degrees of pipelining or serialisation [9]. We are also exploring methods for integrating in our development process the parallel language occam [14], which offers the prospect of allowing the combined hardware and software implementation to be described and verified in a uniform framework. Furthermore, the use of occam will facilitate executing the hardware unit concurrently with the host processor.

Acknowledgements

Thanks to Tony Hoare and Richard Stamper for comments on an earlier draft. The support of Henry Lau, M.P. Luk, Bill McColl, Teddy Wu, Cedric Yiu, Rank Xerox (UK) Limited, Oxford Parallel Applications Centre, Esprit OMI/MAP and OMI/HORN projects, Scottish Enterprise and Algotronix Limited is gratefully acknowledged.

References

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

[2] P. Bertin, D. Roncin and J. Vuillemin, "Introduction to programmable active memories," in *Systolic Array Processors*, J.V. McCanny, J. McWhirter and E.E. Swartzlander Jr. (eds.), Prentice Hall, 1989, pp. 301–309.

[3] D.A. Buell, "Sorting on Splash 2," Technical Report SRC–TR–92–078, SRC, Maryland, September 1992.

[4] B. Fagin and J.G. Watt, "FPGA and rapid prototyping technology use in a special-purpose computer for molecular genetics," *Proc. ICCD*, 1992, pp. 496–501.

[5] M. Gokhale et. al., "Building and using a highly parallel programmable logic array," *IEEE Computer*, vol. 24, 1991, pp. 81–89.

[6] G. Jones and M. Sheeran, "Circuit design in Ruby," in *Formal Methods for VLSI Design*, J. Staunstrup (ed.), North-Holland, 1990, pp. 13–70.

[7] 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.

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

[9] W. Luk, "Systematic serialisation of array-based architectures," *Integration, the VLSI Journal*, Vol. 14, No. 3, February 1993, pp. 333-360.

[10] W. Luk and G. Brown, "A systolic LRU processor and its top-down development," *Science of Computer Programming*, vol. 15, no. 2–3, 1990, pp. 217–233.

[11] W. Luk and G. Jones, "The derivation of regular synchronous circuits," in *Proc. International Conference on Systolic Arrays*, K. Bromley, S. Y. Kung and E. Swartzlander (eds.), IEEE Computer Society Press, 1988, pp. 305–314.

[12] W. Luk, G. Jones and M. Sheeran, "Computer-based tools for regular array design," in *Systolic Array Processors*, J.V. McCanny, J. McWhirter and E.E. Swartzlander Jr. (eds.), Prentice Hall, 1989, pp. 589–598.

[13] 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.

[14] 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.

[15] W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1988.

[16] R. Sedgewick, *Algorithms*, Second Edition, Addison Wesley, 1988.