ACCELERATING SORTING WITH RECONFIGURABLE HARDWARE

Ney L. V. Calazans, Fernando G. Moraes,
Katherine B. Quintans, Felipe B. Neuwald

Faculdade de Informática – PUCRS
Av. Ipiranga, 6681 – Porto Alegre – CEP: 90619-900
{calazans, moraes, quintans, fbarp}@inf.pucrs.br

Abstract: This paper is dedicated to explore the acceleration of sorting algorithms with reconfigurable hardware. We present the rationale for solving the sorting problem in hardware, and suggest ways to ease the use of sorting hardware in the real world of applications programming. One of the ongoing work main goals is the migration of the quicksort algorithm to hardware. The algorithm and its mapping to hardware are discussed.

Keywords: sorting, VHDL, FPGA, digital systems, fast prototyping.

1 Introduction

Ordering data according to some criterion is one of the most basic tasks using in processing information [1]. Although this ordering may be a goal in itself, the main reason for ordering data is mostly to enhance performance of other processing tasks. The data classification problem is usually stated formally using a set theory. We use the designation *sorting* for the theoretical problem associated to ordering.

Problem Statement (Sorting) – *Suppose given $S=\{s_1, s_2, s_3, ..., s_n,\}$, a set of* $\mathbf{n}$ *elements and a sequence of* $\mathbf{p}$ *elements Q $[t_1, t_2, t_3, ...,t_p]$, where* $\mathbf{n \leq p}$*, and each t_i is an elements s_i of S, with any number of repetitions of element of S allowed in Q. Suppose also that there is a total order binary relation < defined over the set S. The* sorting problem *consists in generating a new sequence $SS=[t_1', t_2', t_3', ...,t_p']$, where each t_i' corresponds to exactly one element t_j of the sequence Q, and additionally, the ordering < holds for any pair of elements of SS, i.e. $t_i'<t_{i+1}'$ for i=1, 2, 3, ...,* $\mathbf{p}$*-1.*

The last 40 years have seen an enormous research effort in developing, implementing and testing sorting algorithms in real life problems [1]. Yet, every month new results are published suggesting enhancements in previously published sorting algorithms, or problem specific solutions that enhance performance of some real life applications needing to use sorting techniques. This work investigates strategies for accelerating sorting using reconfigurable hardware platforms and presents some preliminary design data on implementing a sequential sorting algorithm in hardware.

The paper is organized as follows. Section 2 justifies the approach and proposes a framework for allowing the use of the developed hardware sorting mechanisms with ordinary programming techniques. Section 3 explores one efficient sequential sorting algorithm, called quicksort. Next, Section 4 depicts the mapping of this algorithm to hardware. Section 5 describes the hardware prototyping environment. Finally, Section 6 gives some conclusions and proposes directions for future works.

2 Rationale for Hardware Sorting

The formal sorting problem statement does not require any specific ordering criterion to be used while sorting data. There are two commonly employed strategies to make sorting general, i.e. not depending upon a given total order binary relation:

1. Associate one or more integer *keys* with every single data to be sorted, so that the algorithm works upon the data keys only, assuming the natural order of integers as the total order relation <. Additionally, a binary information can be used to parameterize the algorithm for performing sort either in ascending or descending order.
2. Provide the user of sorting with means to externally specify a comparison procedure that, given two elements to sort, decides the relative ordering between them. The sorting algorithm is expected to call this procedure whenever needed.

The second strategy is the most flexible, but we will assume the first one, once the second does not easily fit into hardware implementations and most real life databases rely upon key indexing as the primary means of classification. Also, in the first strategy the size of the key is not related to the size of the data to order, only to the number of records in the database which may reduce memory needs of sorting.

Sorting is a memory hungry computational task. The biggest possible amount of data records should reside in main memory during the algorithm execution. It has been reported that the maximum amount of records that can be sorted into a 256Mbytes Unix workstation is around 1.5 millions [2], in a database indexed by a single integer key of 10 bytes and a pointer to the record on disk. Memory usage considers the operating system (OS) memory overhead, around 38-17% in practical cases. The exact overhead depends on the total memory, measured with 64-256Mbytes. Around 20% of the time to run the sort algorithm is due to CPU time, when using quicksort [3].

The approach proposed here is based on two technological advances. First, there is a trend to standardize the communication between the main processor and peripherals in Personal Computers and/or UNIX Workstations. A good example is the PCI bus standard. In our approach, one or more peripherals house hardware implementations of sorting, sharing access to resources with the processor, typically main memory and disks. The general architecture we assume implies 3 blocks: processor/main memory, hardware accelerator, and secondary storage sharing a bus. A second technological advance is the availability of low cost reconfigurable hardware platforms with high memory capacity and high communication bandwidth with the main processor. Most of these are built with SRAM-based FPGAs.

Let us investigate one example platform. It is possible to acquire a board with capacity to simultaneously support 8 instances of our quicksort hardware implementation plus 16Mbytes of memory for data and a communication bandwidth with the main processor driven by a 32bit/33MHz PCI bus interface. The cost of such a platform is around US$850.00. More expensive platforms allow multiplying hardware by a factor as big as 20, memory by as much as 16 and bandwidth by as much as 4.

This is to be compared with the cost of current systems used to break the barriers of sorting runtime, Symmetric Multiprocessors (SMP) and Network of Workstations (NOWs). In [2], the authors investigate the sorting of lots of data and compare the cost-benefit figures of several systems, costing around US$1 million and sorting 6Gbytes in one minute. We are comparing our approach to these and other results in the literature.

The long-term goal of our work is this field is the development of such algorithms in hardware, integrating implementations with a software library that can be used in ordinary programming. The library is expected to comprise the code to configure an attached hardware platform to run any of these algorithms in hardware on demand. Static configuration means hardware configuration at program loading time, while dynamic configuration means runtime configuration, upon call of the algorithm.

3 The Quicksort Algorithm

Quicksort is one of the most efficient existing sorting algorithms. In the recursive version of the algorithm, described in Figure 1 as pseudocode, the implementation is composed by two procedures, named `QuickSort` and `Partition`. The functionality consists in choosing a certain element, called `pivot` and partitioning the input vector of integer keys into two vectors. The first one contains all elements smaller than or equal to the pivot, while the second contains all elements greater than the pivot. This process is recursively repeated. Figure 1 shows also the non-recursive version of quicksort.

```
void QuickSort(int *v, int p, int r) {
int q;
  if (p < r) {
     q = Partition(v,p,r);
     QuickSort(v,p,q);
     QuickSort(v,q+1,r);
  }
}


int Partition(int *v, int p, int r) {
int pivot, i, j;
  pivot = v[p];
  i = p-1;
  j = r+1;
  while TRUE {
    repeat j = j-1 until v[j] <= pivot;
    repeat i = i+1 until v[i] >= pivot;
    if (i < j)
      exchange  v[i] ↔ v[j];
    else
      return j;
  }
}
```

```
#define MAXSTACK 128 /* Size of the stack */
void QuickSort(int *v, int tamanho){
  int i, j;
  struct limits_type newlimits;
  struct stack_type stack;
  stack.top = -1;
  newlimits.left = 0;
  newlimits.right = size-1;
  push(&stack, &newlimits);
  /* Repeat while some unsorted subvector on stack */
  while(!empty(&stack)){
   pop(&stack, &newlimits);
   while(newlimits.left > newlimits.right){/*next subvector*/
     j = Partition(v, newlimits.left, newlimits.right);
     if(j-newlimits.left > newlimits.right-j){
      /* put lower subvector in the stack */
      i = newlimits.right;
      newlimits.right = j-1;
      push(&stack, &newlimits); /* upper subvector */
      newlimits.left = j+1;
      newlimits.right = i; }
     else { /* put upper subvector in the stack */
      i = newlimits.left;
      newlimits.left = j+1;
      push(&stack, &newlimits); /* lower subvector */
      newlimits.left = i;
      newlimits.right = j-1; } } } }/* end of QuickSort*/
```

Figure 1 – Recursive and non-recursive versions of quicksort.

The behavioral dependence of the `Partition` function affects asymptotic performance in a significant way, since the worst case for it (totally unbalanced partitioning) determines a complexity of $O(n^2)$ for quicksort with an input size of ***n*** elements. On the other hand the best case for the partitioning function dictates a complexity of $O(n \log n)$. Fortunately, it can be demonstrated that the average case time (the most probable in practical problems) presents the same complexity as the best case and not of the worst case [4]. To obtain higher performance, we must eliminate recursion totally, simulating it by the explicit use of stack structures and substituting recursion by iteration.

The non-recursive version works as follows: after the `Partition` function finishes execution, the actual parameters of `QuickSort` are no longer needed, except to compute the arguments for the following two recursive calls. Thus, instead of stacking the parameters implicitly at each recursive call, we can explicitly compute and stack the new actual parameters for each one of the two next calls of `Partition`. In this case, we create an explicit stack that contains the ends of each subvector that still need to be sorted. We stack the bigger subvector and process the smaller subvector immediately after iterating. The choice of processing smaller vectors first is a heuristic to reduce the needs for allocating memory for the stack. Hardware quicksort requires the use of the non-recursive version as basis for the implementation. This occurs because no hardware mechanism is available for implementing recursion.

4 Quicksort Hardware Design

We have implemented and tested a quicksort core in synthesizable VHDL. The design is synchronous. The quicksort core we implemented is divided into three main blocks: **Partition**, **Quick Datapath** and a finite state machine controller playing the role of a **Control Unit**. The external ports of the design include a memory interface. The memory stores both, the vector under sorting and the control stack. Besides the ususal control signals there is a signal to inform the external world sorting has finished.

The **Control Unit** and **Quick Datapath** blocks correspond to the QuickSort procedure. Although we have tried several structural configurations for implementing this part of the design, its sequential structure leads to a datapath structure that is not much different from a conventional small-scale CISC processor. The **Control Unit** is a finite state machine (FSM) with 37 states. The **Partition** block implementation is also in the form of a datapath and a control FSM. However, the structure is simpler and more amenable to parallel actions. The partition **Control FSM** comprises 20 states.

The algorithm implemented by the **Partition** block **Control FSM** is almost exactly the one described in Figure 1. Some differences occur in the treatment of limits of the vector, due to subtle distinctions between the recursive and non-recursive versions.

The synthesis of our implementation (together with the external controller to be described in the next Section) fits in a single 10,000 gate FPGA. It consumes 63% of the logic resources of the device, and an estimated size of 5,000 equivalent logic gates. An analysis of the quicksort algorithm reveals that in the worst case the stack can grow as much as the size of the vector under sorting, although this is a very rarely found situation in practice. Our first design decision was then to locate the stack in memory only, which has the disadvantage of slowing down the algorithm execution and taking up half of the memory space available. The advantage is on the generality of the implementation, which will never run out of stack space. Better solutions can be devised, by a careful dimensioning of the stack (eventually inserting it in the core, if the stack is small enough) and an associated mechanism to allow the stack to grow as needed with a graceful decay in performance. Implementing the stack inside the reconfigurable hardware would have a great impact on the performance of the algorithm, once roughly half of the memory accesses to addresses would be unnecessary. However, if the stack is big, this can affect the hardware resources.

The implemented quicksort core has been designed in order to allow the easy retargeting of the hardware to changing specifications of data and address width. By simply changing two lines in the original VHDL description and resynthesizing the hardware, a new working version is obtained. The size of the hardware version is limited only by available reconfigurable hardware resources, which can be scaled as well during synthesis, by simply choosing a device with more hardware resources.

5 Hardware Prototyping of Quicksort

The first version of the hardware quicksort was done in a standalone hardware prototyping platform and design environment. The interface between the reconfigurable hardware and the memory consists in an 8-bit data bus and a 15-bit address bus, plus three controls signals (chip enable, write enable and output enable). With only 8 bits for data, the hardware is limited to work with small size keys, instead of the more commonly found multibyte keys [2]. One solution to this problem is to multiplex data reading and writing, with a great impact on performance. To enhance performance, next implementations will make use of PCI hardware prototyping platforms, with parallel paths between the reconfigurable hardware and the processor-memory subsystem of either 32 or 64bits. We have implemented a controller to adapt signals from the quicksort core to memory, and to provide feedback on the results of the algorithm execution. The address space of our implementation can be easily expanded to 15 and even to 16 bits (in case of the use of expanded memory versions of the board), but the external controller has to undertake significant changes before dealing with data more than 8 bits long. A first implementation of quicksort was simulated, implemented and debugged.

6 Conclusions and Future Work

Future processors and microprocessors may include reconfigurable logic as part of its internal hardware, to allow greater flexibility and enhanced performance in solving specific problems. By addressing the implementation of hardware tightly coupled to the microprocessor system bus, we expect our approach to constitute a series of experiments on prototyping the internal interface between future in-chip configurable logic and the core processor. Second, our experience showed that the quicksort algorithm is not the most adequate one for hardware implementations, since its nature is intrinsically sequential, offering little place to large-scale parallelism. We are currently starting to investigate parallel sorting algorithms [5] and their hardware implementations. Third, we are currently enhancing the implementation and porting it to a PCI bus pluggable platform to allow more realistic situations and measurements. This involves the development of device drivers and the use of PCI core hardware for high performance communication between the reconfigurable hardware and the host.

7 References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Inc., NJ, 1989.

[2] A. C. Arpaci-Dusseau et alii. High-Performance Sorting on Networks of Workstations. *In: ACM SIGMOD Conference on the Management of Data*, Tucson, AZ, May, 1997.

[3] C. Nyberg et alii. AlphaSort: A RISC Machine Sort. *In: Proceedings of 1994 ACM SIGMOD Conference*, May, 1994.

[4] T. H. Cormen, et alii. *Introduction to algorithms*. Cambridge, MA: The MIT Press, 1990.

[5] R. C. Agarwal. A Super Scalar Sorting Algorithm for RISC Processors. *In: Proceedings of 1996 ACM SIGMOD Conference*, pp. 240-246, June , 1996.