International Carpathian Control Conference ICCC' 2006
Rožnov pod Radhoštěm, CZECH REPUBLIC
May 29-31, 2006

EFFICIENT SORTING ARCHITECTURES IN FPGA

András SZÉLL and Béla FEHÉR
Department of Measurement and Information Systems,
BUTE – Budapest University of Technology and Economics,
Budapest, Hungary, {szell,feher}@mit.bme.hu

Abstract: This paper presents and compares some major hardware and embedded software sorting solutions, with a special care for methods whose implementation can be efficiently done in FPGA. Distributed sorting in a processor network is considered. For specific data sets a special merge-sorting architecture is presented, that consists of block RAM based merge sorters fed by a sorting network. Measurements and comparisons to other existing solutions, especially to sorting performance of PCs are also discussed in the paper.

Key words: sorting, FPGA, merge sort, parallel comparison

1 Introduction: scope and purpose of embedded sorting

Sorting algorithms are a very well known and thoroughly analyzed part of computer science. Many methods and architectures have been discovered and optimized, yet for specific input data in a special environment, the algorithms that are usually meant to be 'optimal' (quick sort and heap sort among others) are not the best solution.

For FPGA-accelerated algorithms, sorting is sometimes an intermediate step for following computations, and in-place sorting is necessary if the communication costs are too high or if an external PC-based sorting is not a possible solution. FPGAs have handicaps in the domain of sorting algorithms against current desktop computers: the smaller memory and slower clock speed, the lack of fast L1 and L2 caches makes sorting generally less cost-efficient, especially if data can be accessed only sequentially from the memory.

One important factor was to achieve the performance of desktop computers for a well defined type of data set. The target architecture was a Xilinx Virtex-II 6000, thus some of the presented sorting solutions rely on its hardware resources.

2 Sorting architectures

For the most effective resource utilization, different sorting methods may be combined together. Such combination is the *std::sort()* method of C++ STL: it is an *introsort* implementation that combines *quicksort* and *heapsort*, giving a worst-case runtime of $O(n \log n)$. This is an asymptotically optimal solution of comparison based sorting on a von Neumann architecture computer. [Musser]

For such complex algorithms a hardware implementation is not viable, still there is a possibility to outperform these algorithms with less sophisticated solutions utilizing the parallelization possibilities of the hardware. *Merge sort* is more hardware-friendly, it is a basic sorting algorithm with $O(n \log n)$ computational complexity. Merge-sorting N elements is done by sorting its two halves separately, and then merging the two sorted parts with N-1 comparisons.

2.1 Sorting network

A sorting network is a simple pipelined architecture of *compare and swap* units organized in a way that the parallel input data gets sorted as it is streamed through the pipeline.

This architecture is optimal for high input data rate, where only small (≤32) sets have to be sorted, as its hardware complexity is $O(n (\log n)^2)$ comparators for n input elements. The network sorts n elements in every clock cycle. [Lang]

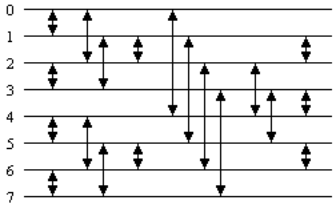

Figure 1. 6-stage sorting network for n=8

2.2 Systolic processing arrays

Systolic processing arrays are simple sorting units connected to their neighbours. Elements are compared and transferred to the neighbours in a way that sorting is done simultaneously with the load of new elements. After data is loaded, the sorted output is ready to be read out from the processing array. The array connections can either be linear or two dimensional, resulting in different resource demands and different performance. A good systolic processing solution is presented in [Bednara].

2.3 Embedded processors

Soft core processors in FPGA like PicoBlaze or MicroBlaze give a good opportunity for parallel sorting, as they can be easily connected to form a processor network.

These processors run software sorting algorithms, and they are best if the comparison itself is a complex operation. In this case there is no possibility to have hundreds of comparators due to hardware limitations, so brute force hardware implementations are not possible. As the comparison operation is expensive, complex software implementations with small comparison counts are better. The operator itself can be partially or fully implemented in hardware and accessed as an extended instruction (e.g. via FSL in MicroBlaze), giving a performance boost for the otherwise slow 50-100 MHz soft core processor. If multiple processors fit in the resource budget, their sorted output can be merge-sorted, either with a specific merger hardware, or with a higher performance embedded hard core processor.

The main bottleneck of sorting in soft core processor hierarchies is the block RAM size of FPGAs and the slow speed of these cores in comparison to PC based solutions. The embedded memory is a limited resource, and program code also consumes it (the MicroBlaze implementation of standard C qsort() routine consumes 4K of memory, and there is only 324K of BRAM memory available in a Virtex-II 6000, which is a considerably big FPGA). The available memory limits the maximum size of sorted data blocks.

2.4 Mixed architectures

If an algorithm works in the FPGA's block RAMs it is bounded by size limits. The possible block size of sorted elements is too small, so many rounds of merge sort has to be performed. For this purpose [Bednara] shows the usage of microprocessors as final merger units; instead of that solution a more resource effective hardware-based merge sorter unit will be described.

3 A special merge sorter

Merge sort works with previously sorted sub-arrays, and is a good choice for sorting separately sorted chunks. The algorithm is very simple and appropriate for hardware implementation. It needs $O(n)$ size of memory for temporary operations; as FPGAs have relatively small memory, this memory consumption is a big disadvantage, but it gives the possibility of further parallelisation as shown later.

The architecture presented here generates small scale sorted blocks and then combines them in a few steps to form 512K x 64 bit sorted output.

3.1 Basic sorter block

This unit is based on insertion sort and has the capability to merge 32 sorted blocks simultaneously. The sorting architecture relies on a multi-level version of this merge sorter.

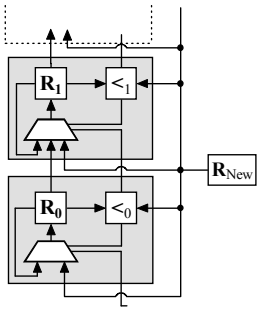

Figure 2. Basic sorter block (2 cells shown)

The basic building blocks of the merge sorter are on Figure 2. Each cell is connected to its two neighbours and to the input data R_{New}. The cells shift up smaller R register values to their next neighbour, and store the R_{New} value where it fits. The equation $R_i > R_{i+1}$ is true in every time instance. The truth table according to the result of the comparison is the following (The comparator output *1* stands for 'R_{New} is bigger'):

Table 1. comparison results and R_1 cell actions

$<_0$	$<_1$	**action**
1	1	$R_1 <= R_0$ (shift)
0	1	$R_1 <= R_{New}$ (insert)
0	0	keep R_1

As a result, the smallest element will eventually reach R_k, the last sorting cell. After that, in every clock cycle the actual smallest element will be shifted to R_k.

These cells are very similar to a systolic processing array, a key difference is that R_{New} is carried to every node separately, decreasing the amount of necessary registers. However it must be noted that because the fan-out of R_{New} has been considerably increased, therefore the routing delay is also longer. This means that larger cell number (e.g. sorting 64 elements) reduces maximal operating frequency.

This simple unit has to be extended in two ways to make it possible to build a full sorter on it. Sorted elements could be read out parallel from the sorter, but that would give a 64 x 32 = 2048 bit data path, which is unnecessary, so for sequential reading, a shifting mechanism is implemented: the sorted elements are shifted out while the empty sorting units take part one by one in sorting the next set of elements, as described in 3.2.

The second extension makes the sorter block capable of merging previously sorted sets from block RAMs.

3.2 Sorter with shift/sort ('active') flag

A state variable ('active' flag) is added to each sorting cell. Inactive cells shift their content regardless of the input, and in every cycle one more cell is set to be active; when the last cell becomes active (and thus the 32 cells store a new set of sorted elements), all are inactivated. This way 32 elements can be sorted in 32 cells, without the necessity to wait for the pipeline to be cleaned up, because the previous set of sorted elements is 'protected' by the flag.

3.3 Merge sorter

Merging *n* sorted arrays is done with the help of *n* additional address registers. If an element is found to be the smallest, it has to be picked off from the top of the array, and the next element has to be compared with the smallest ones of all the other arrays.

In the FPGA these arrays are stored in block RAMs. When an array element is picked off the top of an array, the address of the next element is attached to it, so when the element is the smallest among the 32 sorted elements, a memory address will also be read out from the associated address register, which is in turn increased by one. The address gives the position of the next element of the same array in the block RAM.

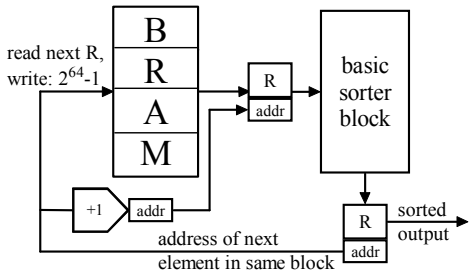

Figure 3. Merge sorter

While the block RAMs are initialized, the sorter block is initialized with the smallest (first) elements of each sorted array. This way the addressing mechanism guarantees that at any time instance the smallest elements of each array are in the sorter block.

For a performance increase, instead of storing which arrays were exhausted during the merge, 2^{64}-1 is written back to the block RAM after each read (dual port block RAMs used with read-first setting, so the read and write are done in one clock cycle). When an address register reaches the end of an array, it overflows, and by reading 2^{64}-1, the new element will be the biggest in the sorting queue, so no 2^{64}-1 element will pop up until all arrays are exhausted and the sorting is finished.

3.4 Proposed architecture

The proposed architecture consist of 256 MB external memory to store the whole input and output array, a smaller 4 MB external memory for parallel temporary storage, a 32-to-1 sorter block with shift/sort ('active') flag; a pair of 1024 x 64 bit block RAMs to store 32 sets of 32 element big sorted arrays; a 32-to-1 merge sorter to generate 32*32=1024 sorted elements; a pair of 16K x 64 bit block RAMs to store these sorted arrays, a 16-to-1 merge sorter that produces 16K sorted blocks from the 16 x 1K blocks and writes it into external memory, and finally, a 32-to-1 merge sorter that reads and merges the temporary external memory back into the 256 MB external memory with the 4 MB of 16K x 64 bit sorted blocks, when it is full.

The pairs of block RAMs are used for parallelisation: while a previous level writes one of the block RAMs, the next level does merge sort on the previously filled part. When they are finished, they swap their working area. This way the inefficient memory utilization of merge sort is solved.

4 Results

Several measurements were done on the input data set. The PC test platform was an AMD Sempron 3000+ on 2 GHz, with 768 MB memory. First a *quick sort* (C++ *qsort()*) implementation was tested, this algorithm sorted the 256 MB input data in 21.0s. Introsort (*std::sort()*) was even faster: 13.61s for simply sorting the 64 bit elements, and 9.72s when a bucket sort preceded the sorting.

The test hardware was a Virtex-II 6000 at speed grade -4. It has an external SDRAM operating at 100 MHz @ 64 bit, its data rate is 6400 Mbps = 800 MB/s. The basic sorter block was implemented with 20 bit comparisons for the same size of sorting keys, operating at a speed over 100 MHz, so it was capable to handle the input data at maximal frequency. Overhead occurs in the following situations:

- during the initialization of the sorter blocks
- when the block RAMs are swapped
- during the last 16K block write, when there is no memory read simultaneously to write

The different overhead factors sum up to a 13% maximal performance loss. Sorting 512K x 64 bit chunks is done in not much more than one full memory read and full memory write cycle, resulting in a 0.64s * 1.13 < 1s performance for sorting 256 MB (32M) of 64 bit elements into 512K blocks.

References

MUSSER, D. R. 1997. Introspective Sorting and Selection Algorithms, *Software − Practice and Experience 27*

LANG, H. W. 2005. Sorting Networks, http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/sortieren.htm

BEDNARA, M., BEYER, O., TEICH, J., WANKA, R. 2000. Hardware-Supported Sorting: Design and Tradeoff Analysis. In *3rd Workshop on System Design Automation*, Paderborn, Germany, 2000, pp 37-44.