MULTILEVEL MODELS FOR DATA PROCESSING

Valery Sklyarov
DETI/IEETA,
University of Aveiro,
Aveiro, Portugal
skl@ua.pt

Iouliia Skliarova
DETI/IEETA,
University of Aveiro,
Aveiro, Portugal
iouliia@ua.pt

Dmitri Mihhailov
Computer Dept.,
TUT,
Tallinn, Estonia
d.mihhailov@ttu.ee

Alexander Sudnitson
Computer Dept.,
TUT,
Tallinn, Estonia
alsu@cc.ttu.ee

ABSTRACT

The paper suggests multilevel models for data processing and demonstrates advantages of such models on examples of data sorting. Three different techniques are discussed, namely graph walk, using tree-like structures and sorting networks. The relevant implementations were done on the basis of hierarchical finite state machines and verified in commercially available FPGAs. Experiments and comparisons demonstrate that the results enable the performance of processing for different types of data to be increased compared to known implementations.

Index Terms— Algorithms; Data models; Data processing; Field programmable gate arrays; Trees

1. INTRODUCTION

Using and taking advantage of application-specific circuits in general and FPGA-based accelerators in particular have a long tradition in data processing [1]. A number of research works are targeted to databases that use the potential of advanced hardware architectures. For example, the system [2] solves a sorting problem over multiple hardware shading units achieving parallelization through using SIMD operations on GPU processors. The algorithms [3,4,5] are very similar in nature, but target SIMD instruction sets of PowerPC 970MP, Cell, and Intel Core 2 Quad processors. The use of FPGAs was studied within projects [6,7] implementing traditional CPU tasks on programmable hardware. In [8] FPGAs are used as co-processors in Altix supercomputer to accelerate XML filtering. The advantages of customized hardware as a database coprocessor are investigated in different publications (*e.g.* [1]).

Some of traditional techniques to data processing are also attractive options in the context of hardware [1]. However, any particular technique cannot be seen as a universal approach producing an optimal result for any set of data. Let us consider, for example, data sorting. There are many methods [9,10] that permit sorting problems to be solved. However, effectiveness of these methods depends on given data. For instance, sorting networks [9] can be seen as one of the fastest techniques but they can be used just for very limited number of data items [1]; tree-like structures [10] are very appropriate for long data that might require fast resorting, etc. This paper suggests combining different models in a way that makes possible to apply particular models at those levels where the selected models permit to produce the best solutions. The effectiveness of such approach was examined for data sorting through combining three different models, namely: 1) the use of the walk technique proposed in [11] at the upper level; 2) tree-like structure for sorting [12] at the middle level; and 3) sorting networks [1] at the bottom level. Obviously, it is not necessary to use all these methods. If the number of data items is limited (for instance, less than 2^8) then just the model (3) allows a nearly optimal solution to be produced. If fast resorting is required, then tree-like structures can be applied autonomously. The upper level provides significant assistance for data with large bit capacity and so on. The most important feature is the ease of linking different models enabling fast data processing for very large volumes of data to be achieved. Further, we will consider data sorting based on the mentioned above multilevel models as an example, which will be analyzed through the paper.

The remainder of this paper is organized in three sections. Section 2 suggests new technique for data processing with examples of sorting. Section 3 is dedicated to implementations, experiments, and the results. The conclusion is given in Section 4.

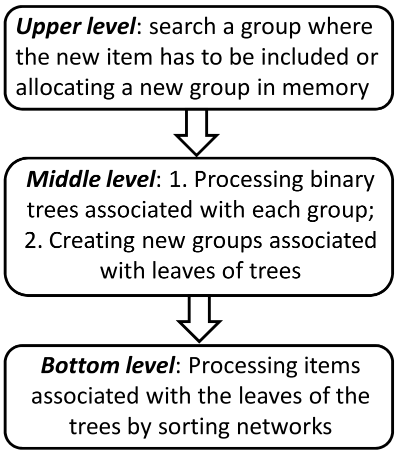
2. MULTILEVEL DATA PROCESSING

Figure 1 demonstrates basic ideas of the proposed multilevel data processing on an example of data sorting. Each particular level is the most appropriate for the following types of data:

- Upper level – for $2^m >> NDA$, where m is the size of data items in bits and NDA is the number of data items;
- Middle level – for data that cannot be processed by the upper and the bottom levels, i.e. this level can be used either autonomously or in combinations with the other methods. Tree-like structures are the most appropriate for autonomous use in applications that

require very fast resorting in case of arriving additional data items;

- Bottom level – as a rule for n ≤ 8 or even smaller, which depends on available resources, where n is the selected number of the less significant bits of data items.



Upper level: search a group where the new item has to be included or allocating a new group in memory
Middle level: 1. Processing binary trees associated with each group; 2. Creating new groups associated with leaves of trees
Bottom level: Processing items associated with the leaves of the trees by sorting networks

Figure 1. Multilevel data processing on an example of data sorting

2.1. Tree walk technique

Suppose we need to sort m-bit data items. Let us decompose m-bit binary values in 2^N groups that differ by the N most significant bits. Let us assume that data items within each group will be sorted by applying some methods, which we will discuss a bit later. At the upper level we have to find a proper group for any new incoming data item. In practical cases for data with large bit capacity (for instance, more than 64) the number of groups is significantly less than 2^N. Thus, we can apply the technique [11] proposed for accelerating Boolean constraint propagation needed for solving the SAT problem with large number of variables and limited number of clauses.

Figure 2 demonstrates an example, taken from [11], in which we consider groups instead of clauses. Similar to [11] the group index walk module uses a tree to efficiently locate the group associated with any new incoming data item. The tree will be N/k deep and it is stored in the tree walk table in an on-chip BRAM block. Here we consider 2^k-nary trees and N/k is the number of assigned levels of the tree, through which the walk procedure from the root of the tree to a leaf (associated with a particular group) will be done. If k=1 we have a binary tree. The greater the number k the less number of levels of the tree we need for our walk procedure. Given a non-leaf node, the address of its leftmost child in the tree walk table is called the base index of this tree node. The rest of the children are ordered sequentially, following the leftmost child. Therefore, to locate the i_{th} child, the index can be calculated by adding i to the base index. If a child is not associated with any group, we store a no-match (-1) tag in the entry. If for a node, all of its 2^k children have no match, then we do not expand the tree node and just store a no-match tag in the node itself. The considered technique is almost the same as in [11].

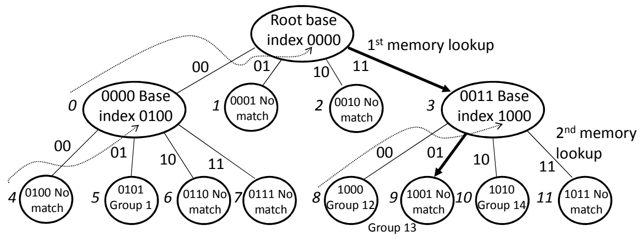


Figure 2. Group index tree work [11] transformed to data sort

Let us consider example from [11] shown in Figure 2 and transform it to the new problem (i.e. sorting will be considered instead of the SAT). Suppose k=2, N=4 and some data items have already been included in the groups 1, 12 and 14. Let the most significant bits of a new data item be 1101. The base index of the root node is 0000 and the first two bits of the data are 11. The table index is the sum of two: 0000+11= 0011. Using this table index, the first memory lookup is conducted by checking the 0011 entry of the table. This entry shows that the next lookup is an internal tree node with the base index 1000. Following this base index, adding it to the next two bits of the input 01, we reach the leaf node 1000+01 = 1001. This leaf node corresponds to the group in which the item 1101 has to be added. Thus, the *no match* index will be replaced with the new group, i.e. group $13_{10} = 1101_2$. New group is indicated by the address of the root of the relevant tree in memory. Now m-N most significant bits of data for groups (such as the group 13 for our example) will be sorted using tree-like structures.

2.2. Tree-like structures

Tree-like data structures are frequently explored for data processing [10,13]. Let us consider an example of using a binary tree for sorting data [13]. Suppose that the nodes of the tree contain three fields: a pointer to the left child node, a pointer to the right child node, and a value (*e.g.* an integer or a pointer to a string). The nodes are maintained so that at any node, the left sub-tree only contains values that are less than the value at the node, and the right sub-tree contains only values that are greater. Such a tree can easily be built and traversed either iteratively or recursively. Taking into account the experiments in [14,15] we can conclude that hardware implementations of recursive and iterative algorithms over trees give very comparable results in resource consumption and execution time. We will base our technique on recursive algorithms just because they provide more clear and compact specification. The results of the paper are equally applied to both recursive and iterative implementations.

Two (following each other) algorithms A_1 and A_2 have to be executed for data sort in [13]. The first one A_1 constructs the tree. The second algorithm A_2 outputs the sorted data from the tree. A_1 executes the following steps: 1) compare the new data item with the value of the root node to determine whether it should be placed in the sub-tree headed by the left node or the right node; 2) check for the presence of the node selected by 1) and if it is absent, create and insert a new node for the data item and end the process; 3) otherwise, repeat 1) and 2) with the selected node as the root. Recursion can easily be applied in point 3 [15]. Let us assume that a sequence of input data is the following: 24, 17, 35, 30, 8, 61, 12, 18, 1, 25, 10, 15, 19, 20, 21, 40, 9, 7, 11, 16, 50. A tree built for this sequence is shown in Figure 3a. Basic steps of A_2 are shown in Fig. 3b. We will assume that input data are stored in RAM along with the addresses of the left (LA) and right (RA) sub-trees (see Figure 3c). All other details can be found in [14,15].

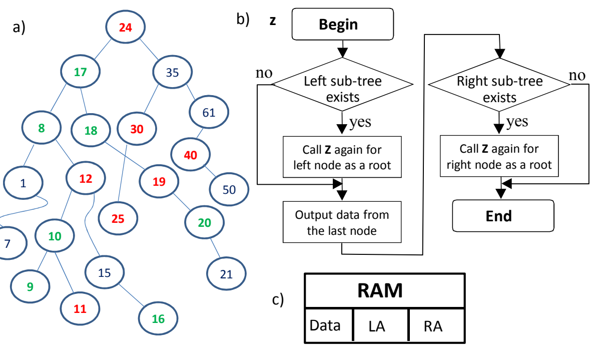


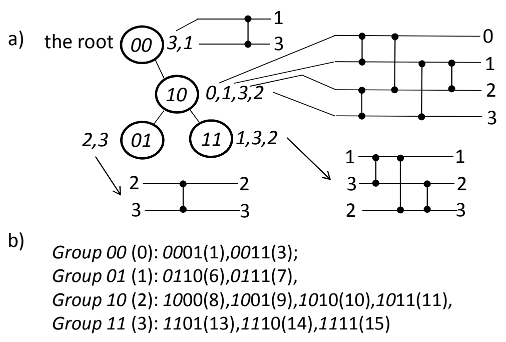
Figure 3. Binary tree for data sort (a); recursive algorithm for data sort (b); contents of memory (c)

2.3. Sorting networks

Sorting networks [1] do not require control flow of instructions or branches and they are parallel in nature. However, they are only suitable for relatively short sequences of data whose length is known a priori [1].

This paper suggests combining tree-like structures (see the middle level in Figure 1) with sorting networks (see the bottom level in Figure 1).

Suppose we need to sort m-bit data items. Let us sort (m-n) the most significant bits at upper two levels (see Figure 1) and process the remaining n bits using sorting networks. Let us consider an example shown in Figure 4 for m=4, n=2 and the following sequence of inputs: 3(0011), 8(1000), 1(0001), 6(0110), 9(1001), 13(1101), 15(1111), 7(0111), 11(1011), 10(1010), 14(1110), where the binary code of the relevant decimal value is shown in parenthesis.



Group 00 (0): *00*01(1),*00*11(3);
Group 01 (1): *01*10(6),*01*11(7),
Group 10 (2): *10*00(8),*10*01(9),*10*10(10),*10*11(11),
Group 11 (3): *11*01(13),*11*10(14),*11*11(15)

Figure 4. Combining tree-like structure and sorting networks

For this example m-n=2 the most significant bits arrive in the following sequence: 00, 10, 01, 11. The relevant bits are shown in *italic* in Figure 4 and in the text. The tree for this sequence is shown in Figure 4a. There are two less significant (n=2)-bit values associated with the group *00* namely 11 (3) and 01 (1). The values 3 and 1 are written near the node *00* in Figure 4a. By analogy (n=2)-bit values associated with each group are shown near the other nodes *10*, *01* and *11* in Figure 4a. All associated values are ordered in ascending sequence using sorting networks (see Figure 4a where networks from [1] were used). Thus, the groups are sorted using tree-like structures and the values associated with the groups are sorted using networks. Comparators needed for the networks are represented in Figure 4a through the known Knuth notation [9]. The results of sorting are shown in Figure 4b.

Note that it is not necessary to store the associated with the nodes values explicitly. They can be indicated by values '1' of bits in the respective binary word. Such words for the nodes *00*, *10*, *01* and *11* in Figure 4a are 0101, 1111, 0011 and 0111.

3. IMPLEMENTATIONS AND EXPERIMENTS

The considered methods were implemented and tested in FPGA. Processing of trees at upper and middle levels was done with the aid of a hierarchical finite state machine (HFSM) [16] using the methods [15]. Specifications for the algorithms (*e.g.* Figure 3b) can be seen as flow-charts with some predefined constraints. Such flow-charts can easily be converted to a HFSM (using the method [16]) and then formally coded in a hardware description language such as VHDL. The coding is done using the template proposed in [14], which is easily customizable for the given set of flow-charts. The resulting (customized) VHDL code is synthesizable and permits the relevant circuits to be designed in commercially available CAD systems, such as Xilinx ISE (all necessary details can be found in [14-16]).

The synthesis and implementation of the circuits from the specification in VHDL were done in Xilinx ISE 11 for FPGA Spartan3E-1200E-FG320 of Xilinx available on NEXYS-2 prototyping board of Digilent Inc. Since the experiments were performed in a cheap FPGA that has limited resources the methods used at each level were verified separately. Preliminary results for the upper level are very similar to [11] because the method [11] was used with minimal modifications. For the middle level a random-number generator produced up to 2^{12} data items with a length of 14 bits that are supplied in portions to the circuits that have to sort the previously received portions of data and to resort them for a new portion as soon as possible. Besides, the both types were evaluated without and with sorting networks for different values of n.

Fig. 5 permits to compare acceleration of resorting for a new portion (that includes from 10 to 120 data items) comparing with resorting all 2^{12} data items in FPGA-based circuits.

The algorithm shown in Fig. 3b was also described in C++ and implemented in software. The same data (randomly generated) were used for the software implementations. The results were produced on HP EliteBook 2730p (Intel Core 2 Duo CPU, 1.87 GHz) computer. They show that hardware implementations are faster than software implementations for all the experiments even though the clock frequencies of the FPGA and the PC differ significantly. Indeed, the clock frequency of the PC is about 25 times faster than that of the FPGA. However, in spite of a significantly lower clock frequency, the performance of sorting operations in FPGA is about 4.5 times faster than in software implementations. This is because the optimization technique [12,16] valid just for hardware circuits has been applied.

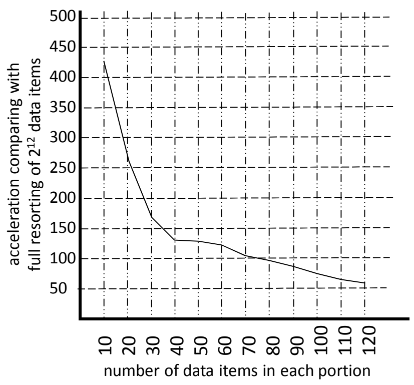


Figure 5. The results of experiments

When we combined tree-like structures and sorting networks we were able to provide acceleration of sorting in an average 1.6/3.1/4.7 times for different values of n: n=2/n=3/n=4 respectively. Regardless of parallelization of 2^n operations, acceleration cannot be equal to 2^n because the use of sorting networks gives significant delay in getting the results due to long paths in the relevant combinational circuits.

Now let us make a comparison with other known results. In [1] the complete median operator to process 256 MB of data consisting of 4-byte words takes 6.173 seconds, i.e. 100 ns per word. Any resorting for a new portion of data requires all data items to be sorted and, thus, for data from Figure 5, it takes about 4 milliseconds. For our technique and for examples in Figure 5, it takes from 130 to 1100 nanoseconds for different number of items in portions.

In [17] the maximum speed of sorting is estimated as 180 million records per second. Thus, resorting all data in Figure 5 would require about 23 milliseconds.

4. CONCLUSION

The paper suggests multilevel models for data processing and clearly demonstrates the advantages of the innovations proposed based on prototyping in FPGA and experiments with the implemented algorithms. The technique is illustrated by examples of data sorting where three different models (graph walk, tree-like structures and sorting networks) are combined.

5. REFERENCES

[1] R. Mueller, J. Teubner, G. Alonso, "Data processing on FPGAs", Proc. VLDB Endowment 2(1), 2009.

[2] N.K. Govindaraju, J. Gray, R. Kumar, D. Manocha, "GPUTeraSort: High performance graphics co-processor sorting for large database management", Proc. 2006 ACM SIGMOD Int'l Conference on Management of Data, Chicago, IL, USA, pp. 325-336, 2006.

[3] H. Inoue, T. Moriyama, H. Komatsu, T. Nakatani, "AA-Sort: A new parallel sorting algorithm for multi-core SIMD processors", Proc. Int'l Conference on Parallel Architecture and Compilation Techniques (PACT), Brasov, Romania, pp. 189-198, 2007.

[4] B. Gedik, R.R. Bordawekar, P.S. Yu, "CellSort: High performance sorting on the Cell processor", Proc. 33rd Int'l Conference on Very Large Data Bases (VLDB), Vienna, Austria, pp. 1286-1297, 2007.

[5] J. Chhugani, A.D. Nguyen, V.W. Lee, W. Macy, M. Hagog, Y.K. Chen, A. Baransi, S. Kumar, P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture", Proc VLDB Endowment 1(2), pp. 1313-1324, 2008.

[6] D.J. Greaves, S. Singh, "Kiwi: Synthesis of FPGA circuits from parallel programs", Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2008.

[7] S.S. Huang, A. Hormati, D.F. Bacon, R. Rabbah, "Liquid Metal: Object-oriented programming across the hardware/software boundary", European Conference on Object-Oriented Programming, Paphos, Cyprus, 2008.

[8] A. Mitra, M.R. Vieira, P. Bakalov, V.J. Tsotras, W. Najjar, "Boosting XML Filtering through a scalable FPGA-based architecture", Proc. Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA, 2009.

[9] D.E. Knuth, *The Art of Computer Programming*, Volume 3: Sorting and Searching, 2nd edn., Addison-Wesley, 1998.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stain, *Introduction to Algorithms*, 2nd edition, MIT Press, 2002.

[11] J.D. Davis, Z. Tan, F. Yu, L. Zhang, "A practical reconfigurable hardware accelerator for Boolean satisfiability solvers", Proc. of the 45th ACM/IEEE Design Automation Conference - DAC 2008, pp. 780 – 785.

[12] V. Sklyarov, I. Skliarova, "Recursive and Iterative Algorithms for N-ary Search Problems", Proc. of AISPP'2006, 19th IFIP World Computer Congress - WCC'2006, Santiago de Chile, Chile, August 2006, pp. 81-90.

[13] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[14] V. Sklyarov, "FPGA-based implementation of recursive algorithms," Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs, vol. 28/5-6, pp. 197–211, 2004.

[15] D.Mihhailov, V.Sklyarov, I.Skliarova, A.Sudnitson. "Hardware Implementation of Recursive Algorithms". In Proc. of the 53rd IEEE Int. Symposium on Circuits and Systems, Seattle, USA, August, 2010.

[16] Sklyarov, V, Hierarchical Finite-State Machines and Their Use for Digital Control, IEEE Transactions on VLSI Systems, 1999, Vol. 7, No 2, pp. 222-228.

[17] R.D. Chamberlain, N. Ganesan, "Sorting on Architecturally Diverse Computer Systems", Proc. 3rd Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications, November 2009.