Computing and Informatics, Vol. 32, 2013, 859–876

THE ENHANCEMENT OF A COMPUTER SYSTEM FOR SORTING CAPABILITIES USING FPGA CUSTOM ARCHITECTURE

Paweł Russek, Kazimierz Wiatr

AGH University of Science and Technology
Department of Electronics
Mickiewicza 30
30-059 Cracow, POLAND
&
ACC Cyfronet AGH
Nawojka 11
30-950 Cracow, POLAND

e-mail: {russek, wiatr}@agh.edu.pl

Abstract. The primary goal of the presented experiment was to judge the usefulness of FPGA technology in the sorting operation performed by computer systems. We were interested to see if it was possible to achieve better system performance and lower energy consumption when the CPU is supported by FPGA chips. The method of custom processing was applied. We proposed dedicated sorting hardware to increase performance and save energy. Our concept addresses High Throughput Computing (HTC) systems. The custom hardware approach was proposed because this technique is available in supercomputing infrastructures today. Another important issue of the work is that the hardware was programmed using High Level Language (HLL). As a semiconductor platform for hardware implementation the FPGA was chosen. We evaluated the efficiency of an FPGA based sorting processor that was programmed in Mitrion-C HLL. The FPGA approach was compared to the CPU approach in terms of efficiency and power consumption.

Keywords: Custom computing processors, data sorting, reconfigurable systems

1 INTRODUCTION

Typically, data sorting is not considered a problem that is suitable for hardware acceleration. Along with other data mining algorithms which are rather bandwidth-intensive than computationally-intensive, a sort operation is not recognized as a problem which requires a dedicated processor. Opinions presented in the literature state that custom processors can offer substantial acceleration only for computationally-exhaustive algorithms. It is also well known that computer architecture that is efficient for sorting must offer efficient data transfer between different levels of memory hierarchy. It is recognized that for a sorting computer system, high computation power has no significant influence on the overall system performance. The computational complexity of the sorting is rather small ($O(n \log n)$). The algorithms that are successfully executed by dedicated processors should be complex such as matrix-multiplication ($O(n^3)$) for example. The above arguments indicate that fast data transfer is more essential than computing power for the sorting systems' efficiency and that considering hardware acceleration is pointless.

Despite that, in this paper we would like to focus on a problem of processing element efficiency. We want to compare the common CPU approach to a novel FPGA solution. Presumably, the presented results might lead to an idea of building a multi-FPGA system in the future. In such a system a part of the CPUs could be replaced by FPGA chips to get better performance and efficiency. Here, in this paper, we compared the CPU approach to the FPGA method. We judged and emphasized the advantages of the FPGA based computational method.

In our opinion, the efficient computational element dedicated to sorting is important if today's available transfer bandwidth is taken into consideration. A lack of computational power can affect the overall sorting performance of a computer system. As an example, let us consider the data throughput rate of $T = 6\,\mathrm{Gbit/s}$ which is available even on home PCs as a new generation of the SATA bus emerged today. For such bandwidths, if we assume eight bytes of data for a single database record, it is possible to transfer $W \approx 100\,\mathrm{M}$ records of data per second. To sort this quantity of records it is necessary to perform more than $W * lgW = 2.6\,\mathrm{G}$ 'compare and swap' operations per second. This requires a substantial computational effort, which is significant even for the fastest processors today. According to our experiments, the state-of-the-art CPUs can perform ca. 500 million of 'compare and swap' operations per second which applies to data that is stored in a processor's cache memory. It is worth mentioning that for professional storage systems used in data and computing centers even $T = 6\,\mathrm{GBytes/s}$ are available today.

Custom designed processors offer an advantage of instruction level parallelism. Both pipelining and consecutive execution are inferred. As only the necessary functional elements are implemented we can gain better performance for the equivalent silicon area. This also means that it is possible to have more computational power for the same system cost. The method reduces system flexibility as sepa-

rate semiconductor devices must be used for different tasks but this problem can be inhibited if reconfigurable hardware such as the FPGA is incorporated. Recently, reconfigurable logic devices (such as FPGAs) have become one of the available platforms for High Performance Computing (HPC). The technique of their utilization is usually referred to as High Performance Reconfigurable Computing (HPRC).

The dedicated processors are also regarded as more energy efficient than CPUs. A well known silicon device power formula is $P = CV^2f$, where C is constant for the certain semiconductor technology, V is power supply voltage, f is a clock frequency. For the FPGA, the clock frequencies are typically 10 times lower than the clock frequencies of the CPUs. This explains why custom FPGA solutions are regarded as green computing architectures that consume less energy. This issue has currently become very important. It was published in [20] that a typical Google Web search request which is sent from a desktop computer to Google's servers, generates about seven grams of carbon dioxide. This makes two searches comparable to bringing a tea kettle to boil. The sorting operation is essential for every database engine so despite the correctness of the above claim, green computing is undoubtedly a forthcoming challenge.

When the programming of reconfigurable hardware is discussed, one of the most important limitations that should be considered is the design effort. Usually, the hardware design is much more time consuming and complex than the designing of software. This generates additional costs of the solution. This expense must be compensated for by a decent speed-up rate that can be gained. This is the reason why acceleration methods that involve hardware customization are often rejected. This holds particularly when no substantial speed-up is noticed. Usually, designers regard speed-up values that are satisfactory to be 10 times or more.

The effort of hardware design can be reduced in various ways. One of the methods is utilization of High Level Languages (HLL). Although this method usually causes performance reduction, it can be treated as a first approach that allows to get early results very fast. If the results are promising, the very next stage can be redesigning of the architecture using Register Transfer Level (RTL) description. This allows to gain maximum performance.

The outline of this paper is as follows. In Section 2 we present hardware implementations of the sorting algorithms that we found in the literature. In this section we also give arguments to choose a certain methodology and describe how the problem was approached. In Section 3 we introduce the reconfigurable platform that was used in our experiment. In Section 4 we explain the theoretical concept of the implemented architecture. In Section 5 we explain the software application used for the sake of comparison with the hardware. In Section 6 we give the implementation results in terms of hardware resources and the performance of hardware and software solutions. We also provide some energy consumption considerations there. In Section 7 we summarize our work and give some perspectives for further research.

2 THE GOAL OF THE EXPERIMENT

Most of the solutions that are presented in literature with regard to procesors dedicated for sorting are solutions for embedded systems. For example, a serial sorter chip is presented in [2]. It shows a very interesting, cost effective idea which is limited in practice to the input data of a moderate size. In this solution all sorted elements must fit in the sorter registers. The sorter is serially fed with data which are consequently shifted toward the sorter output. During their way through the sorter registers, elements are serially compared to one another. Although the solution is very simple and economical it is rather more suitable for embedded systems and cannot be considered for high performance architecture. The sorter was implemented in 2 μm CMOS semiconductor technology.

Similar to the previous is the [3] work. It proposes a chip of the continuous sort. It consists of serially connected sorting elements. Each sorting element is coupled with a register to store single list elements. It performs a compare operation of stored and newly arriving elements and passes the smallest one to the next sorting element. The length of the sorted list is limited by the number of sorting elements in a queue. This number was 450 for the presented VLSI implementation. The sorter throughput is 50 MWords/s. Each data record consisted of 24 bits of key and 48 bits of data.

In [4] an efficient, parallel sorting architecture, denoted as MM-SORT (min-max sort) is regarded. The proposed architecture took advantages of a sorting network and distributed processors coupled with local memory. In this solution a small sorting network can be used iteratively to sort a large number of elements. The major upside of the idea is that it can be executed in pipeline if some conditions are fulfilled. There is no need to wait for the results of the previous iteration to start the next one. A large size distributed memory is necessary in this solution. This holds because the complete input data set must fit simultaneously into the sorting processor's memory. The authors did not implement their proposal in practice so no results are known.

Both a merge sorting and Batcher's sorting network are discussed in [5]. In this method a list of elements is first partitioned into shorter lists. Batcher's network is used to sort those lists. Short lists are then merged into a final list by the merge sorting. This solution does not have high performance characteristics as the merge sorting is performed sequentially for elements which are stored in local memory.

In [6], the Burrows-Weeler transform for data compression is considered. In the work enhanced wavesorter architecture is proposed. It is a parallel processing module suitable for sorting of relatively short sequences (100 elements as mentioned in the paper). To exploit its efficiency, a wide parallel input port is required. This limits the usage of the architecture as a processor for HTC because it requires excessive bandwidth to saturate performance.

In [7] different sort architectures for embedded systems are discussed. Three different architectures are evaluated. All presented solutions are intended for the FPGAs. They are implementations of: Batcher's sorting network, insertion sorting

and merge sorting. Merge sorting combined with insertion sorting was found to be the fastest one. Depending on the number of elements, the speed-ups between 1.5 and 16 that were achieved in comparison to the quick sorting algorithm that was running in software.

Beside embedded designs, high performance solutions can also be found in the literature. The FPGA implementation is described in [8]. The distinction of this solution is that it was designed for real time video application. The performance of the solution was the key factor. The disadvantage of this proposal was the lack of universality. A Counting-Sort algorithm was implemented so the input data set had to fulfill certain conditions. The solution was capable of sorting 128 keys in a shorter time than 1μs. It outperformed other VLSI solutions. However, no comparisons to the CPU-based solutions were presented.

A hybrid hardware/software solution of a sorter is considered in [9]. It uses merge sorting for its sequential (software) phase, and Parhami's and Kwai's [10] systolic insertion sort for the parallel (hardware) phase. It is contrary to the solution presented in our paper where Merge-Sort is performed in hardware.

The solution presented here is intended for HTC problems. The primary goal is to achieve the maximum throughput of the system. It can be used for the acceleration of database engines for instance. The issues and challenges related to data storage strategies that arise today are discussed in [11]. The feasibility of an FPGA approach was studied in [12]. Bandi et al. explained and presented the mechanisms of hardware acceleration of database queries.

A study of the above presented literature leads to the conclusion that hardware acceleration of sorting has its limitations. It is applicable rather for embedded systems where system performance per cost is the most important constraint. Solutions that are incorporated for embedded systems do not fit HTC needs. If we focus on the performance of a solution, algorithms of smaller numerical complexity, i.e. Heap-Sort and Quick-Sort, must be rejected as sequential in nature. No instruction level parallelism could be exploited in the structure of a hardware processor.

Quick-Sort and Heap-Sort can be executed much faster by the CPUs than by the FPGA custom processors. However, we recognized Bubble-Sort, Bucket-Sort and Merge-Sort as algorithms which offer some advantages for parallelism. As pipelining is the strongest feature of hardware acceleration we further investigated the algorithms for adequate features. Finally we recognized the best potential was in the Merge-Sort algorithm and we decided to focus our interest on it.

The Merge-Sort algorithm is simple and easy to parallelise. Its computational complexity is the same as that of the Quick-Sort and Heap-Sort algorithms. The Merge-Sort algorithm is widely used in practice to build highest performance sort engines. When it comes to practical software implementation, Merge-Sort can outperform other algorithms due to its consecutive data access. The Merge-Sort algorithm starts with unsorted data but it can operate also with already sorted lists. For best performance, it must be combined with other algorithms. As a result, the fastest sort solutions combine Quick-Sort or Heap-Sort algorithms with a Merge-Sort algorithm [13]. In general, the sorting scheme is as follows: first short lists

are sorted by a Quick-Sort or Heap-Sort algorithm and then they are merged by a Merge-Sort.

The idea developed here is to use the CPU to sort lists whose size fits into the processor cache and then finish sorting by Merge-Sort performed in the dedicated hardware. A Merge-Sort processor can be used to speed-up intensive merge sorting operations in any software application. This idea has a proper background in the cited references [13]. They prove that the best results were achieved by hybrid solutions. We will compare that of Merge-Sort implemented in FPGAs with that of a Merge-Sort application. If the hardware implementation of Merge-Sort is faster than the software one then the overall system performance of the hybrid hardware/software solution will be better than the performance of the solely software solution.

In our experiment, we first designed and then optimized the processor architecture suitable for merge sorting. Thus, two different sorting architectures are presented and explained. The first one takes advantages of flip-flop registers as storage elements. The second one uses RAM memory to store intermediate data. Both were compared in terms of performance and resources utilization. As it was primarily an experiment in the area of the FPGA based data mining, we leveraged designing process by utilization of Mitrion-C [14] as a design entry tool. We accepted a moderate loss of hardware performance as a cost of lower designing complexity. Thanks to that we were able to evaluate our sorting processor idea faster. The architecture was run on an FPGA based reconfigurable hardware platform SGI RASC RC100 [15]. The RASC is a reconfigurable computing platform which comes with SMP servers of the SGI's Altix 4 700 family. Finally, we compared the performance of an FPGA solution with the speed of an algorithm that runs on the CPU.

It should be noted that the performance comparison was done with the Itanium2 1.5 GHz Intel processor. There are of course many more advanced CPUs available at present. However, the CPU referred to was manufactured in the same 90 nm technology as used and deployed in the Xilinx's Virtex4 FPGA. As semiconductor technology is developing very quickly, and new processors are available in a short period of time, the only way to conduct trustworthy comparisons is to compare the FPGAs and the CPUs of respective semiconductor technology. This was done to maximize the accuracy of our experiment.

3 RECONFIGURABLE COMPUTING PLATFORM

The most important issue when the hardware acceleration platform is considered is the fast and efficient data transfer between the system memory and the hardware accelerator or the hardware accelerator cache memory (depends on architecture).

In a PC based reconfigurable computing systems the FPGA accelerators were attached to the computer through peripheral bus like PCI or PCI-X bus for instance. Such a solution has limitations which keep down good acceleration results. In such a case, the processor disposes of a very fast local system bus with instant access

to the system memory, unlike the reconfigurable accelerator that has slow memory access. Thus, hardware acceleration is not so attractive because the data transfer time from the system memory must be added to the hardware processing time. Even if the CPU itself performs slower than the FPGA it can be recognized as a better solution because of the quick data transfer. In computationally intensive applications the FPGA must be tightly integrated with the rest of the computing system.

There is another approach implemented in the HPC systems where the FPGA is treated as an integral rather than peripheral system component. It is linked directly to a processor's resources through high-speed connections and so overcomes the biggest bottleneck of the FPGA co-processing. To provide maximum performance, an FPGA co-processor has the same memory access capabilities as the CPU. For example, in a Cray XD1 reconfigurable system, the FPGA is integrated with the processor system bus. Respectively, the SGI Corp. proposed an adequate solution in its Altix family systems. It has already proved its usability in HPC applications [16]. Altix is a family of SGI's SMP (Symmetric Multi Processing) solutions. It is based on the Itanium2 1.5 GHz processor. Its distinguishing feature is that each processor has both fast access to its local memory and slower access to local memories of the other processors. The Inter-processor data is achieved by a relatively fast bus – NUMALink. The NUMALink interconnect is a hierarchical system bus. It allows global addressing and scalability of the SMP system. The maximum NUMALink data transfer is 6,4 GB/s. The integral component of the Altix system can be the Reconfigurable Application Specific Computing (RASC) module. It enables users to develop application specific hardware using reconfigurable logic elements. The SGI RASC is tightly integrated with NUMALink. From the hardware perspective the FPGA is no longer in the co-processor mode in this model. The NUMALink allows the FPGA to access the global shared memory and there is no need to load and unload data. The RASC is coupled with two Virtex4LX200 FPGA chips [17]. Each one offers 200 k of reconfigurable logic cells. Additionally there are two blocks of 40 MB QDR RAM memory. This memory acts like a second level cache for the FPGA. The first level cache is implemented inside the Virtex4LX200 structure and is called BlockRAM. The bidirectional data interface implemented for the FPGA has 128-bit width and is timed with a clock frequency of 200 MHz.

4 SORTER ARCHITECTURE

The term 'architecture' seems to be opposite to what was stated in the previous section that the HLL was used to create the design. Typically, in a HLL design flow a device is created according to a so called high level behavioral description. This usually means that instead of a structural hardware description, i.e. 'architecture', a behavioral source code is used. On the other hand, the intended hardware behavior always has its corresponding hardware structure. If a programmer understands the philosophy of the HLL tool used then s/he can predict what hardware will

be inferred as a result of certain HLL behavioral statements. This means that programming of architecture using HLL is also possible. Additionally, we believe that for better results, a designer should be aware of the final hardware structure. This should be kept in mind during behavioral programming. It is different in software programming. Even if the HLL languages and software languages have similar or the same syntax. Unfortunately, software and hardware programming have nothing in common. Despite the fact that the Mitrion-C leverages design process by handling many design aspects it cannot create the hardware itself. First the architecture should be devised and then it can be described using the Mitrion-C syntax. Here we present the intended architecture that was programmed later in the HLL. We will present diagrams of the architecture to make the description more comprehensive. Contrary to the architecture block diagrams, the HLL's source codes are not provided in this paper. If it is required the reader can obtain them from the web site [18].

4.1 A Sort Tree

The basic outline of the presented architecture is a binary tree, called the sort tree in this paper. The tree is presented in Figure 1. It consists of storage and control elements. The functionality of the control element is presented in Figure 2. It is a kind of comparator/multiplexer device. It compares data on DATA.IN.LEFT and DATA.IN.RIGHT input ports and feeds the smaller value to the DATA.OUT output.

There are also LEFT.GT.RIGHT and NOT LEFT.GT.RIGHT control outputs. They pass the results of the comparisons from the root of the tree to the leaves and this activates the left or right path above the particular node. The DATA.OUT is latched by an output storage element when ENABLE signal is active. At each moment only one of the paths from the root to the top leaves is active. The stored data records are shifted down the tree along a selected active path.

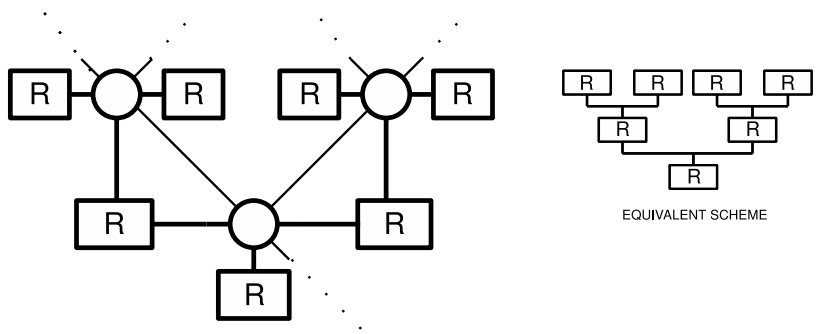

Fig. 1. The basic structure of the sort tree

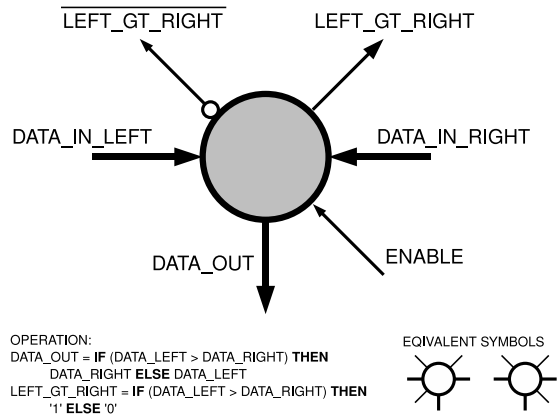

Fig. 2. The control element of the sort tree

An example of the Sort-Merge system is presented on Figure 3. In addition to the sort tree, two memory banks are necessary. The input memory bank stores lists that are to be merged. In the case of an eight fold merging system the 'list 0' is always loaded to R8, 'list 1' to R9 and so forth. The lists are emptied according to the index of the storage element which is active at the moment. For example, if R8 is shifted the next data from 'list 0' is read from the main memory and fed into R7. Data from R1 are stored in consecutive locations of the output memory. The last element of each input list is INFINITE symbol to avoid a list underflow.

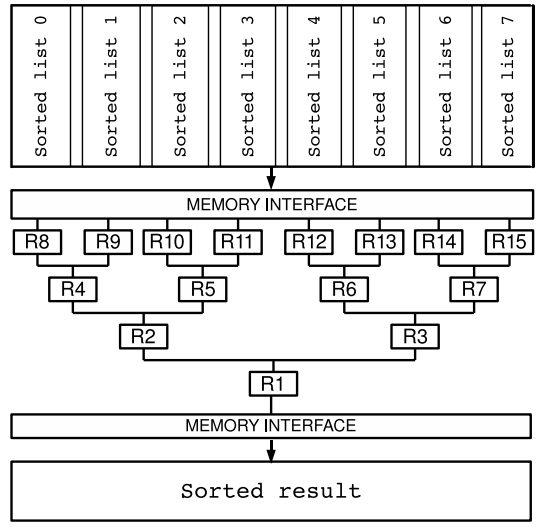

Fig. 3. The sort tree composed of eight elements

4.2 The Sort Tree Implementations

Two different versions of the sort tree were implemented. They differ in the type of FPGA resources that were used to implement storage elements. In the first version of implementation, flip-flop registers were used to store data. This directly implements the introduced sort tree concept. In the second version, data records were stored in the FPGA internal memory blocks called BRAMs (Block RAMs). The second version requires some additional control logic as the generation of the RAM address is necessary. Later in this paper the corresponding versions are called register (reg) and BlockRAM (mem).

As the register version is a direct implementation of the sort tree, no additional explanation of this architecture is necessary. Now, the BlockRAM version will be examined closer. The concept of this architecture was to spare flip-flop resources of the FPGA and utilize the BlockRAMs (BRAMs for short). The usage of the BRAMs are recommended when the implementation of extended memory elements are necessary in the FPGA. Although it is a little bit cumbersome to employ the memory blocks in Mitrion-C, we were able to implement the BRAM version of our architecture.

The architecture concept is presented in Figure 4. It is an eight-fold version of Merge-Sorter. Sorted elements are kept in corresponding memory cells. The storage elements that are located at the same level of the sorting tree share the same memory block. There is also a distinction for left and right child storage elements. They are separated in the different memory blocks. Consequently, it is possible to achieve simultaneous access to both children for the particular tree node. The addresses of the appropriate elements in each memory block (i.e. activation of the proper memory elements on each level of the tree) are generated according to the comparison results from the lower parts of the tree. This is similar to the register based version despite the fact that the values must be read from the memory. From a single node point of view, the elements of smaller value are read from the memory of the higher level and then it is written to the memory of the lower level. The highest level memory blocks are endowed by lists' elements stored in the RASC main memory. This top/down transfer is performed at each algorithm step.

The advantage of the BRAM based architecture is that it consumes significantly less of the FPGA resources. This usually enhances better performance of the solution e.g. increasing the number of parallel modules. On the other hand, the additional control logic is more complicated which can be a performance degradation factor.

As an HLL was used, the design process was not very time consuming. The design time is a very important factor when a practical solution is considered. In our case by employing the Mitrion-C it was possible to achieve a very short design time. It took approximately 32 hours of designer's time to code and simulate the architecture in an HLL. In addition, approximately 6 hours of computer work was necessary to synthesize and place & route automatically each single implementation. The reported design time refers to the first architecture of the particular version. The

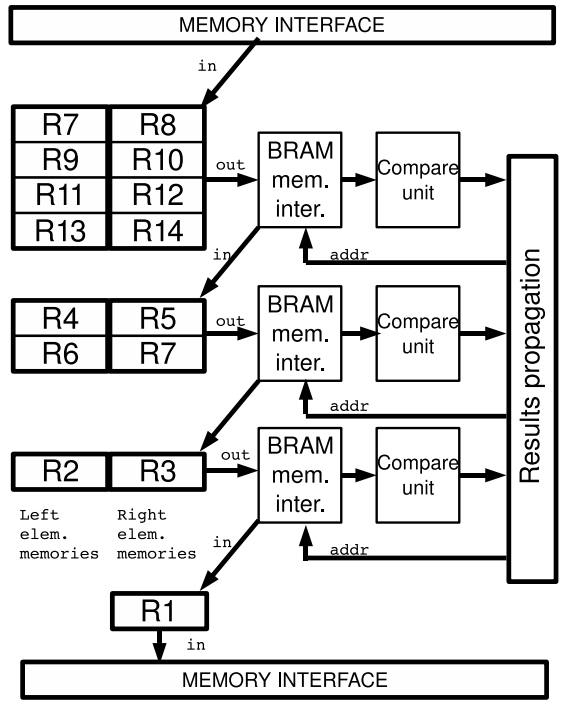

Fig. 4. The Block RAM version of the sorting tree

next implementations that differ only in generic parameters were gained instantly after additional synthesis and the place & route process.

The sorted lists consist of elements treated as records. The single record has a KEY field and a DATA field. Records are sorted according to the KEY field values. Because of the data width of the external memory offered by the RASC, the implemented record size is 128 bits. The size of the KEY and DATA varies, but the total size of the KEY and the DATA must be 128 bits.

The main generic parameters of the design that are used in the Mitrion-C source code are as follows:

NUM.OF.STR parameter which determines number of lists to be sorted,

SEGMENT.LEN parameter that determines the number of records in a list,

ELEM.TOTAL is a total number of all elements,

TYPE.KEY parameter that defines the bit size of the KEY part of a record,

TYPE.DATA parameter which defines the bit size of the $DATA$ part of a record.

5 SOFTWARE APPLICATION

To compare performances of hardware and software solutions, two applications were written. The first took advantage of the sort tree hardware. The second was a pure software solution. In both cases the goal was to sort records stored in the form of lists in the dynamically allocated memory. The total size of the data set was 16 MBytes. The result was written to another dynamically allocated memory.

In the case of the hardware based solution, the allocated memory data was sent to the input memory bank of the RASC platform. Then, the hardware algorithm was launched. After the algorithm completion, the results were read from the output memory of the RASC.

For the purpose of performance comparison, the Merge-Sort application was developed. A pseudo code for the executed algorithm is presented in listing 1.

Algorithm 1 Sort $NUM.OF.STR$ lists (L) using Merge-Sort algorithm. Store result into M. $M \leftarrow MergeSort(L[\text{NUM-OF-STR}])$

```
Lists ⇐ L
Nbr ⇐ NBR.OF.STR
Len ← SEGMENT.LEN
Total ← ELEM.TOTAL
{Allocate memory for data structures}
Allocate Heap[N] {for heap}
Allocate Index[N] {for heap's index}
Allocate Result[Total] {for sorted data}
{Create new list from the first elements of the lists. Create index}
for i = 1 to Nbr do
   Heap[i] ← L[N][0]
end for
Heap ← Build-Heap(Heap)
Index ← CreateIndexForHeap(First, Heap)
{Iterate all elements in the lists}
for t = 1 to Total do
   Result[t] ← Heap[0];
   index = Index[0]
   Heap[0] ←next element from L[index]
   Heap ← Heapify(Heap)) {Restore heap property in Heap}
   Heap ← UpdateIndex(Index) {Update Index}
end for
M ← Result
```

The software algorithm starts with data structures allocation. The memory for Heap, Index and Result is reserved. Then, the Heap of the first elements of the lists is created along with the Index to identify the lists that correspond to the particular Heap element. The Heap is created according to the Build-Heap algorithm described

in [19]. Iteration starts when the top element in the heap is picked up as the smallest one and stored in the Result memory at the next available position. From the list that corresponds to the just picked up element the consecutive element is read and stored as the heap top. To allow the next iteration, heap property of the Heap must be restored. This is done according Heapify algorithm presented in [19]. After NBR_OF_STR × SEGMENT_LEN iterations all elements in the lists are fetched.

6 THE EXPERIMENT AND RESULTS

To perform the Megre-Sort experiment, sorted lists were generated. The lists consisted of 128 bits records. Each record was split into two fields: KEY field and DATA field. The sorting was performed according to the KEY field. The DATA field did not affect the result sequence. The size of the KEY and the DATA fields was different for the register based Merge-Sorter and for the memory based one. Because of better memory availability, 64 bits KEY and 64 bits DATA could be used for the memory based Merge-Sorter. In contrast to the register based Merge-Sorter, only the 32-bit key and the 96-bit data were implemented. The implementation of the 64-bit key was not feasible for the register based architecture because in this case the routing tool did not meet the timing requirements. That was due to too high resource utilization.

The presented algorithms differ in NBR.OF.LISTS and the tree type. The type of the merge tree can be memory (mem) or register (reg) based. For all algorithms the total number of records is 1 million (16M Bytes of data). That size corresponds to the size of the input memory block of the reconfigurable computing module. All algorithms names and types are listed in Table 1. The sort procedure was performed for the same set of records both in software and hardware. The performance results are given in Table 2. The table also includes the FPGA utilization data which is measured in the number of used slices. The slice is a basic logic element of the Xilinx FPGA matrix. It consists of Look-Up tables (LUT) and flip-flops (FF). In Table 3 detailed hardware utilization is reported in terms of the number of occupied LUTs and FFs. The number of used Block RAMs is also included. The clock frequency for the Mitrion-C based hardware project is always fixed at 100 MHz.

Alg. name	Alg. type	Nbr. of list	Lists len.	Key size	Data size
reg8	reg	8	0x20000	32	96
reg16	reg	16	0x10000	32	96
reg32	reg	32	0x8000	32	96
mem8	mem	8	0x20000	64	64
mem16	mem	16	0x10000	64	64
mem32	mem	32	0x8000	64	64
mem64	mem	64	0x4000	64	64

Table 1. Parameters of the implemented algorithms

Alg. name	FPGA time	CPU time	Speed up	Slices	Slices [%]
reg8	237 ms	287 ms	1.2	26 009	22 %
reg16	237 ms	342 ms	1.4	29 878	33 %
reg32	236 ms	380 ms	1.6	44 355	49 %
mem8	239 ms	319 ms	1.3	18 012	20 %
mem16	236 ms	340 ms	1.4	20 376	22 %
mem32	237 ms	378 ms	1.6	24 696	27 %
mem64	239 ms	480 ms	2.0	34 545	38 %

Table 2. Performance and resource utilisation of the Merge-Sort implementation

Alg. name	LUTs	LUT [%]	FFs	FF [%]	BRAMs	BRAM [%]
reg8	26 009	12 %	26 009	14 %	44	13 %
reg16	35 004	19 %	36 402	20 %	76	22 %
reg32	52 086	29 %	54 070	30 %	140	41 %
mem8	16 668	9 %	22 767	12 %	35	10 %
mem16	19 288	10 %	25 060	14 %	44	13 %
mem32	24 762	13 %	28 517	16 %	52	15 %
mem64	35 656	19 %	38 491	21 %	60	17 %

Table 3. Detailed resources utilisation of the Merge-Sort implementation

For the sake of energy consumption comparison we employed the Xilinx's power estimator (XPE) to judge the energy consumed by a Virtex4 chip. The power consumption for the FPGAs strongly depends on the particular implementation. It differs for various configurations. We assessed the energy of the architecture of the best performance and parameters, i.e. 'mem64'. For the used resources and the clock frequency it consumes 1.3 W of quiescent and 1.8 W of dynamic energy. The total power is 3.1 W only! We could not measure the exact energy consumption for the Itanium2 chip due to the lack of proper equipment. As we compare chip to chip energy consumption we would have to measure energy directly on the CPU socket. Energy consumption of the CPU depends on the executed code so we can not report it in any strict way here. What we can cite, is the maximum thermal power dissipation (TPD) for the Itanium2 reported by Intel. It is 104 W. It is obvious that difference between the FPGA and the CPU in power consumption is different by an order of magnitude.

7 CONCLUSIONS AND FUTURE WORK

The conducted experiment revealed that the performance of the CPU and the FPGA for the Merge-Sort algorithm are almost similar in practice, when an HLL description is used. There is a slight advantage of the FPGA system. The speed up is more substantial for larger problems, e.g. a speed up of 2.0 was gained when 64 lists are merged simultaneously. The execution time on the FPGA is the same for every problem size. This is because the number of processing elements is proportional to

the size of problem. Thus, better FPGA performance over the CPU for bigger problems could be expected. For the bigger sort tree, the maximum resource utilization was not higher than 49 % for register solution and 39 % for memory solution. There is still some overhead in the FPGA resources which could be utilized to increase the size of the implementation (to 128 lists version for example). However, for k-times larger problems, there is only $lg(k)$ times increase in the calculation complexity. So, there is not much improvement in speed up when the problems are bigger.

As it could be predicted, the FPGA resources utilization is higher for the register architecture. This is due to the utilization of the BRAMs in the memory version of the architecture. In fact, there is no substantial difference in the performance of both the architectures. It could be expected that the memory based architecture might be slower because of higher complexity of the address of the memory generation unit. However, in the HLL implementation the difference between both architectures diminished.

Another important issue worth analyzing is the pipeline latency of our architecture. Let us analyze the problem below.

The elements from the list memory are read ELEM.TOTAL times. We call that process the main algorithm loop. During each loop cycle a necessary comparison and replace operations must be performed. According to the FPGA performance results, it can be estimated that each loop cycle takes approximately 20 clock cycles ($\approx$ clock frequency $\times$ FPGA time $\div$ ELEM.TOTAL). It is quite a lot if we look at the proposed architectures. According to the authors' experience in Hardware Description Languages (HDL) programming, it can be expected that for careful low level implementation it would be possible to use not more than two clock's periods per one loop cycle for the register version and five clocks per cycle for the BlokRAM version. The BRAMs based architecture uses the FPGA's resources in a more efficient way but it is more complicated in the implementation and as a result slower if compared to the register based version. As a consequence of the above, the HDL description would guarantee additional speed-up of 4.0 times and 10 times for each type of architecture.

Although the achieved advantages of the FPGA over the CPU are promising this can offer practical consequences in real computer systems under several circumstances. First of all a single FPGA chip can not compete with the multi node CPU systems. One FPGA chip corresponds to one CPU chip; so to get a substantial advantage for the HTC the number of FPGAs must be comparable to the number of CPUs. However, at present, the price of an FPGA device is prohibitive. The price of reconfigurable systems cause the existing installations to have a status of research platforms rather than production solutions. Dissemination of the FPGAs in computing installations would reduce the costs, but at the moment this technology is recognized as difficult in programming. The tools are still in the early stages of development and they still require further development. So, unfortunately, the practical aspect of our proposal has a very long-term prospect.

It can be concluded that the FPGA could be a useful technology if the speed up of sorting operations is necessary. In terms of power consumption the FPGA is

unbeatable compared to the CPUs. The FPGA with a 100 MHz clock frequency can be seen as a green computing solution in contrast to the 1.5 GHz Itanium2. For the performance purpose, however, it would be necessary to enhance the implementation and rewrite the proposed architectures using some HDL, e.g. VHDL or Verilog.

Acknowledgements

Scholarly work financed through research funds by The National Center for Research and Development (NCBiR) as a SYNAT research project in 2011.

REFERENCES

[1] Leake, J.—Woods, R.: Revealed: The Environmental Impact of Google Searches. The Sunday Times, January 11, 2009.

[2] Afghahi, M.: A 512 16-b Bit-Serial Sorter Chip. IEEE Journal of Solid-State Circuits, Vol. 26, 1991, No. 10, pp. 1452–1457.

[3] Colavita, A. A.—Cicuttin, A.—Fratnik, F.—Capello, G.: SORTCHIP: A LSI Implementation of a Hardware Algorithm for Continuous

[4] Data Sorting. IEEE Journal of Solid-State Circuits, Vol. 38, 2003, No. 6, pp. 1076–1079.

[5] Zhang, Y.—Zheng, S. J.: A Simple and Efficient VLSI Sorting Architecture. Proceedings of the 37th Midwest Symposium on, Circuits and Systems 1994, Vol. 1, pp. 70–73.

[6] Huang, C.-Y.—Yu, G.-J.—Liu, B.-D.: A Hardware Design Approach for Merge-Sorting Network. ISCAS, Vol. 4, 2001, pp. 534–537.

[7] Martinez, J.—Cumplido, R.—Feregrino, C.: An FPGA Parallel Sorting Architecture for the Burrows Wheeler Transform. Proceedings of the 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig '05), p. 17.

[8] Marcelino, R.—Neto, H.—Cardoso, J. M. P.: Sorting Units for FPGA-Based Embedded Systems. DISPES, 2008, pp. 11–22.

[9] Ratnayake, K.—Amer, A.: An FPGA Architecture of Stable-Sorting on a Large Data Volume: Application to Video Signals. 41st Annual Conference on Information Sciences and Systems 2007 (CISS '07), pp. 431–436.

[10] Bednara, M.—Beyer, O.—Teich, J.—Wanka, R.: Trade-Off Analysis and Architecture Design of a Hybrid Hardware/Software Sorter. In: Procweedings of Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP00) 2000, pp. 299–308.

[11] Parhami, B.—Kwai, D. M.: Data-Driven Control Scheme for Linear Arrays: Application to a Stable Insertion Sorter. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, 1999, No. 1, pp. 23–28.

[12] Krol, D.—Funika, W.—Slota, R.—Kitowski, J.: SLA-Oriented SemiAutomatic Management of Data Storage and Applications in Distributed Environment. Computer Science, 2011, pp. 37–50.

[13] Bandi, N.—Sun, C.—Agrawal, D.—El Abbadi, A.: Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations. Proceedings of 30th International Conference on Very Large Data Bases (VLDB '04) 2004, pp. 1021–1032.

[14] Nyberg, C.—Shah, M.: Sort Benchmark Homepage. Availaible on: `http://sortbenchmark.org/`.

[15] Mitrionics AB: Mitrion Users' Guide. Availaible on: `http://www.mitrion.com/`.

[16] SGI, Corp.: SGI RASC RC100 Blade. Availaible on: `http://www.sgi.com/pdfs/3920.pdf`.

[17] Wielgosz, M.—Mazur, G.—Makowski, M.—Jamro, E.—Russek, P.—Wiatr, K.: Analysis of the Basic Implementation Aspects of Hardware-Accelerated Density Functional Theory Calculations. Computing and Informatics, Vol. 29, 2010, No. 6, pp. 989–1000.

[18] Xilinx, Corp.: Virtex-4 User Guide. Availaible on: `http://www.xilinx.com/`.

[19] Russek, P.: Merge-Sorting Source Codes in Mitrion-C. Availaible on: `http://www.fpga.agh.edu.pl/russek/sorting.rar`.

[20] Cormen, T. H.—Leiserson, C. E.—Rivest, R. L.: Introduction to Algorithms. The MIT Press and McGraw-Hill Book Company 1989.

Paweł Russek received M. Sc. degree in Electronics from AGH University of Science and Technology in 1995 and the Ph. D. degree from the same university, in 2003, with research on customized architecture for image compression algorithm in FPGA. He works as an Assistant Professor in the Department of Electronics, AGH UST and as a research fellow in the Academic Computing Center 'Cyfronet' AGH. His research interests focus on hardware acceleration in FPGA. He is particularly interested in high performance reconfigurable computing, embedded systems and the digital systems design based on high level synthesis languages. He is an author and a co-author of many publications in the field.

Kazimierz Wiatr received the M. Sc. and Ph. D. degrees in electrical engineering from the AGH University of Science and Technology, Krakow, Poland, in 1980 and 1987, respectively, and the D. Hab. degree in electronics from the University of Technology of Łódź in 1999. He received the Professor degree in 2001. His research interests include design and performance of dedicated hardware structures and reconfigurable processors employing FPGAs for acceleration computing. He received 9 research grants from the Polish Committee of Science Research. These works resulted in above 200 publications, including 8 books, the most recent one being Acceleration Computing in Video Processing Systems. He is also the author of 5 patents and 35 industrial implementations. He was the reviewer of: IEEE Expert Letters, International Journal Eng. App. of Artificial Intelligence, IEEE Transactions on Neural Networks, Journal Machine Graphics and Vision, Eurasip Journal on Applied Signal Processing. He currently is the Director of Academic Computing Centre CYFORNET AGH, and ithe Head of PIONIER council-Polish Optical Internet.