University of Kaiserslautern
Department of Computer Science
Database and Information Systems

Seminar
Optimizing data management on new hardware

Summer Semester 2014

Table of Contents

MapReduce on FPGAs

Manuel Hoffmann

University of Kaiserslautern

Abstract. After having explored commodity hardware as target platform for big-data computations with MapReduce, the database community has started to look for alternatives. Also the FPGA-community seems to look for applications in the area of big-data computations, trying to adopt the intuitional model of computation of MapReduce.
In this term paper, we will look into approaches that try to consolidate FPGA-based hardware acceleration and MapReduce algorithms that operate on large data sets. We will see two frameworks which try to enable programmers to exploit hardware acceleration in order to execute MapReduce jobs faster, and one cluster that demonstrates the benefits of using FPGAs amongst other processors.
Although we will see the performance increasing by an order of magnitude, such a boost is only made on the concrete cases. Unfortunately, we will neither see a speed up for general computations e.g. handling arbitrary strings nor a system that is targetet to the same experienced developers who write the MapReduce jobs that are today running in the houses of big-data.
Still, we get a valuable insight in the possibilities of using specialized hardware as accelerators and get an understanding of how MapReduce jobs can benefit from them and which efforts have to be made in return.

1 Motivation

When Dean and Ghemawat published their MapReduce paper [DG04] in 2004, they introduced an easy to use framework which enabled programmers to write understandable functions that are automatically applied in parallel to distributed data. In research, MapReduce (in particular the open-source implementation Hadoop) is used as basis for several projects like in [SJD13] where the mentioned functions are replaced by algebraic operators and [MD14] where different sorting algorithms are implemented.

On the other hand, the use of Field Programmable Gate Arrays (FPGAs) in order to accellerate database specific computations was proposed by Mueller and Teubner in [MT09], which might enable having database specific computing machines. An idea that failed in the 70s as it was too costly, but we will see how FPGAs can be configured to be the specific machine that is optimized for our needs.

As control logic is encoded into the circuits, FPGAs do not need overhead instructions for array indexing or loop computations – instead, every operation can "deliver payload" [HVG+07]. So we can expect that FPGAs deliver a great performance when computing specific solutions, especially when we combine FPGAs with the parallel model of MapReduce. This is, what this seminar work is about.

1.1 Structure of the seminar work

In this thesis we will focus on an explanation what FPGAs are from a perspective of information-systems engineers. We compare FPGAs to CPUs, as they are the commonly used machines. Then we shortly revisit MapReduce. The rest of this paper is arranged as follows: section 3 presents some leading thoughts of how to

fuse FPGAs and MapReduce. Section 4 shows efforts of researchers which have already built MapReduce systems that contain FPGAs and section 5 compares the differences in their approaches. Finally, in section 6 we draw conclusions on the whole topic.

2 Fundamentals

2.1 Field Programmable Gate Arrays

FPGAs are micro chips that consist of multiple gates which can be arbitrarily wired and this way "allow tailor-made hardware designs optimized for specific systems, applications, or even user queries" [MT10]. In order to get a better intuition and at the same time a deeper understanding of FPGAs, we consider an exemplary chip architecture in figure 1, there the fundamental components of an FPGA are depicted as taught in [Weh14]. They are:

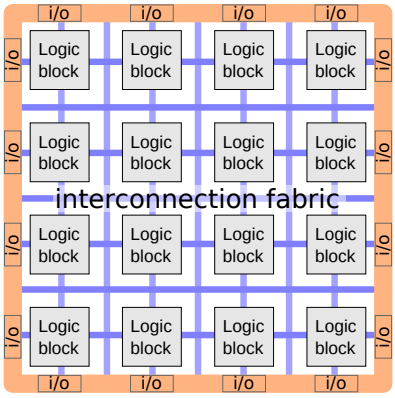

Fig. 1. Simplified structure of an FPGA [Weh14]

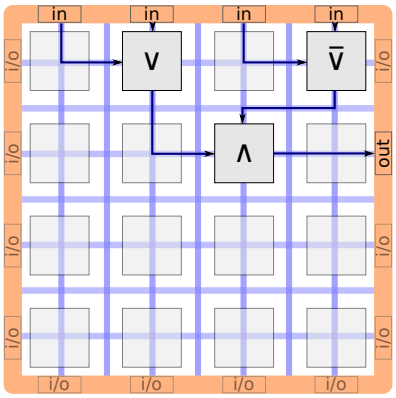

Fig. 2. The same FPGA under a configuration

- several **logic blocks** which basically can emulate any logical function in size of their inputs.
- A ring of **i/o blocks** (input/output) which connect the FPGA to its environment.
- The **interconnection fabric**, a circuit that associates logic blocks and i/o blocks among each other.

Depending on the vendor, FPGAs can also be equipped with additional components like digital signal processor (DSP) cells or local memory [Ges14]. Such an FPGA can be mounted on a board and connected to a system bus like PCIe [TL10]. Or it could be delivered in stand-alone boxes and connected by Ethernet, where a dedicated Ethernet controller is provided [YLH12]. Also these boards and boxes usually have possibilities to mount RAM modules, which provide more locally accessable memory to the FPGA.

As the name indicates, FPGAs are field programmable. This means, the circuit design of an FPGA is not fixed, but is adjustable at will even when the FPGA is deployed ("in the field"). Programming is done by flashing the **configuration** onto the chip. The configuration fixes the function which each logic block computes, mode of i/o blocks and the **routing** between the blocks through the interconnection fabric.

Programming an FPGA Programming an FPGA actually means, generating such a configuration. For exploring the configuration, let us consider the toy algorithm in figure 3 with inputs A, B, C, D, intermediate variables X, Y and output Z (in the fashion of Gessler's introduction to embedded systems, [Ges14] ch. 2).

```
X := A ∨ B;
Y := ¬(C ∨ D);
Z := X ∧ Y
```

Fig. 3. Simple data transforming algorithm

```
LOR $X, $A, $B
LOR $Y, $C, $D
NEG $Y, $Y
AND $Z, $X, $Y
```

Fig. 4. The same algorithm as MIPS-like assembler program

We want this algorithm to be executed on an FPGA, so we have to provide a configuration. This has to map the operators to logic blocks, the inputs and outputs to i/o blocks, and the order in which the operations are applied to the variables to a routing.

Figure 2 shows how the configuration for this example can set up the FPGA: The algorithm is executed by just sending the inputs to the FPGA, and—assuming a synchronous setup where each operation needs one clock cycle—getting the results from the output pin after two cycles. The configuration itself is encoded as a bitstring and has to be sent to the FPGA before the inputs arrive.

Let us compare this to the execution of the same algorithm on a CPU. Instead of a configuration, an instruction stream—assembly code—has to be generated. Each instruction consists of an opcode, telling the CPU which operation to perform, and the addresses of the operands for this particular operation. The algorithm from figure 3 translated to a MIPS-like assembly code can look like the source listing in figure 4.

The assembly version follows the sequential character of the original algorithm. It describes how the computation sequentially transforms one variable into another. Thus this approach is also called *computation in space*. The FPGA on the other side computes different parts of the algorithm on different logic blocks in parallel, so this is called *computation in time* [Ges14].

So the unique selling points of FPGAs that discriminate them against CPUs are the possibility to be configured according to the very specific problem we want to solve and at the same time the ability to compute several aspects in different places in parallel. At least the first point is obviously only a benefit, if the same configuration (i.e. the same algorithm) can be applied many times to different input data.

Practical aspects In practice, neither one wants to directly write assembly code nor generate a configuration by hand, especially if the algorithms are getting bigger. Instead for CPUs there are high-level languages which are compiled down to assembly code (e.g. C++, Haskell, ...) or even are interpreted on the fly by a runtime environment (e.g. Java, Ruby, ...). For FPGAs on the other hand there are hardware description languages (e.g. Verilog, VHDL, ...) which can be used to in advance simulate hardware behaviour and finally generate configurations for specific chips. There is also the approach of "high-level synthesis" where code of a programming language is transformed into a hardware description, but there are reports of non-optimal performance compared to directly development of hardware models [YTT+08], [SWY+10].

What makes the task even more difficult is that "FPGA application performance is unusually sensitive to the implementation's quality" [HVG+07] since Amdahl's law tells us that in order to get a speedup of factor 50, at most 2% of the program is allowed to be sequential.

To extend the previous comparison between FPGAs and CPUs: As a generalist, a CPU is normaly tightly integrated in its computing environment. There is data access to the main memory with caches and interfaces to other components on the hardware side, and on the software side there are well-proven working operating systems that take care of user interaction, networking and so on. Compared to this, a FPGA seems like a blank sheet. All things that one has to take in mind, when talking of using FPGAs in order to make things faster, and so we will mostly see FPGAs running crucial steps of algorithms but with backup of a CPU handling a variety of backgroud tasks.

Real World FPGAs We close this introduction of FPGAs with some numbers that give a better understanding of the situation on the market. Xilinx, the first vendor of FPGAs, ships products that are equipped with 20K–2M logic blocks, 13Mb–68Mb block RAM, and a theoretical i/o bandwith of 6.6–28.05 Gb/s [Xil14]. Another Vendor, Altera, ships their Stratix FPGAs with 80K–4M logic blocks, several technologies for on-board RAM, and i/o rates upto 56 Gb/s [Alt14]. Especially these data rates are enormous e.g. compared to Intel's most recent Core-i7 processor, whose memory bandwith is capped at 25.6 Gb/s [Int14].

2.2 MapReduce

MapReduce is an architecture proposed by Dean and Ghemawat for processing huge amounts of data that is distributed over several nodes in a cluster [DG04]. The two naming core functions are **map** and **reduce**:

- $\text{map} : (k, v) \mapsto (k_I, v_I)^*$
- $\text{reduce} : (k_I, v_I^*) \mapsto (k_O, v_O)^*$

The map function is executed on every node and transforms input key-value pairs (k, v) to several intermediates (k_I, v_I). All intermediate values that belong to the same key k_I are then grouped together (thereby possibly sent over to another node) and processed by the reduce function, which then emits output that is again stored in the cluster.

We will refer with *map* and *reduce* to the according functions. A unit executing these functions is called *mapper* or *reducer* and a program instructing which map and which reduce function is to be applied on which input is known as a *job*. The framework around is simply called *MapReduce* but when appropriate we also will use this term for the overall concept. A widely used open-source variant of this framework is Hadoop [Had14]. It provides several *management* functions, as reading text files and providing these as key-value pairs for the mappers, correct instantiation of mappers and reducers, data transfer between the nodes and writing output pairs into files. Figure 5 destilledly shows the structure of a MapReduce application, where the red dashed lines indicate control of the master program over the worker nodes and the blue arrows show the order of data processing.

As the core idea of MapReduce is to execute simple functions repeatedly on a big-data corpus, this seems to be a perfect model for approaching computations on FPGAs.

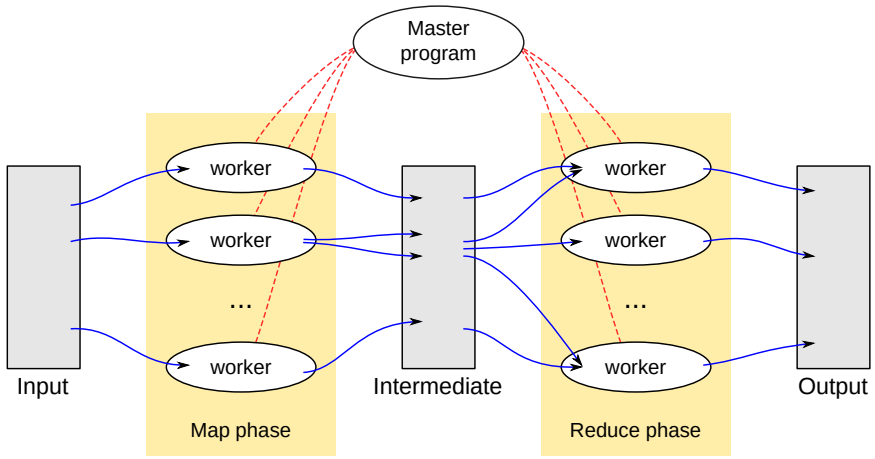

Fig. 5. MapReduce architecture according to [DG04]

3 Possible Applications

Let us consider the basic components of a MapReduce framework: input and output (i/o), management and the mappers and reducers. If we want to distribute the execution among CPU and FPGA, figure 6 shows the probably most intuitive three possibilities: to do only the bare execution of functions on the FPGA (*FPGA-slave*), to also put some management tasks on the chip (*shared-management*) and let some on the CPU, or to do do only i/o on the CPU (*all-FPGA*). Let us now elaborate these possibilities.

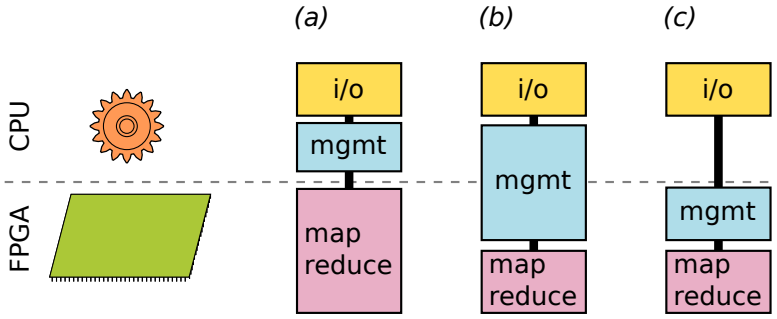

Fig. 6. Possible distributions of the MapReduce components. *(a) FPGA-slave*: management of tasks happens on the CPU, the FPGA is only used for computation, *(b) shared-management*: management tasks happen on both, CPU and FPGA, *(c) all-FPGA*: management and computation tasks are placed on the FPGA

3.1 FPGA-slave

Maybe the simplest way to bring FPGAs into play is to take a MapReduce framework and implement map and reduce functions which send their data to the FPGA, receive the results back and emit them to the MapReduce framework. We call this approach **FPGA-slave** since on the FPGA there is no management functionality located.

For this idea we need to make sure, that the configuration is loaded onto the FPGA before the map and reduce functions are executed. Depending on the capabilities of the framework, we also have to take care of concurrent accesses to the FPGA. For example, the CPU has four cores and on each runs a single-threaded mapper. Then either the FPGA has to have four map modules which can be addressed in parallel, or we need four FPGAs connected to the CPU. Or we need a process scheduler on the host CPU.

Beneficial in this approach is, that the existing framework takes care of distribution of functions and intermediates, and also failure recovery. As we basically have the whole FPGA for one task computation, a downside can be, that we could underutilize its resources by not exploiting enough parallelism.

3.2 Shared-management

The process-scheduler option of the FPGA-slave idea leads us to this one: **shared-management**. Here, the management tasks are split up between CPU and FPGA. The CPU's responsibility is reading input data, and storing intermediate and output data. According to the current phase, it sends the tuples to the FPGA. Now on the FPGA there are several mappers and reducers located. A scheduling module takes care of sending input tuples to a free map or reduce module.

In the FPGA-slave approach, the framework just executed one task at a time, thus one tuple got to the FPGA at a time. Here, the framework can read input data and send a continuous tuple stream to the FPGA faster than the latter can compute results. In order to prevent this tuple overflow, the FPGA must have a feedback channel that is used to communicate its status.

3.3 all-FPGA

The last approach is, to do only i/o on the CPU, sending input to the FPGA and receiving only the computed MapReduce output back. As all computations happen on this chip, we call this approach *all-FPGA*.

Now we need not only mappers and reducers on the FPGA. But also a scheduler like in the shared-management approach and—most severely—we need place to manage and store intermediate values. The last factor seems to be extremly limiting the job size, as we cannot spill intermediates to a hard disk.

Also this way, we scale down MapReduce to just one node, since the intermediate tuples do not leave the FPGA. So this approach is best fitted for jobs that boil down the data size in the map function, and where input data is not distributed over several nodes.

Summary We neglected the idea of doing even i/o on the FPGA. But (a) there is no implementation following this approach and (b) as discussed before, the FPGA is used as a highly specialized circuit, whereas i/o is a task that relies on many different tasks so that the generalist CPU should take care of it.

This is meant to be a discussion of the pure doctrine before we see real works deviating from that in the next section.

4 Examplary Architectures

After having spent some time on thought experiments describing possibilities of how to fuse MapReduce with FPGAs, we now will look into three implementations:

1. The unnamed MapReduce framework by Yin et al., which can be classified as *FPGA-slave*. As they signed their paper "Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware" [YLH12].
2. **Axel**, a cluster built by Tsoi and Luk and described in [TL10]. Besides GPGUs and CPUs it includes FPGAs as computational elements, being a *shared-management* arrangement of MapReduce.
3. The last implementation is called **FPMR**, as mix of FPGA and MapReduce, developed by Shan et al [SWY+10]. FPMR is finally a framework that runs as *all-FPGA* solution.

On every of these three approaches, we will have a look at the architecture, implementations of an example algorithm and appropriate performance measurements. Since the architectures will differ heavily, a direct performance comparison is not impossible.

4.1 Scalable MapReduce Framework

In 2012 Yin et al. presented a framework which allows users to write MapReduce jobs and specify tasks for FPGAs in Verilog [YLH12]. Their overall approach is, to integrate FPGAs into the map and reduce computations of the well-known and open-source MapReduce implementation Hadoop.

Architecture This framework is built upon Hadoop, thus there is a cluster of interconnected worker nodes. Each worker node consists of one CPU and several FPGAs. These workers are assigned tasks, which are divided into *real tasks* and *virtual tasks*: while real tasks are Hadoop tasks that are executed on the CPU, virtual tasks do their actual computations on the FPGA; on the CPU there is just a driver program running, issuing communication with the FPGA.

As showcase let us take a look at the execution of a virtual map task: At the beginning of the map phase, Hadoop's input reader translates input data into initial key-value pairs. The FPGA Worker Driver of the unnamed framework wraps these pairs up in ethernet frames sending them to the FPGA. On the FPGA, the so called Task Selector assigns the incoming pair to an according worker module (several differnet map and reduce worker modules can be located on the same chip). After processing, the resulting tuples are sent back over ethernet to the FPGA Worker Driver and are forwarded to Hadoop's OutputCollector.

We have seen that the FPGAs are connected to the CPU via Ethernet. This way, one can easily attatch more than one FPGA to the CPU. However this introduces the need of doing network management on the FPGA board: before the actual map and reduce functions are executed, the Packet Decomposition module splits up header and payload and sends the payload as described before to the Task Selector. For simplicity, 'the header' means MAC, IP and UDP header and we do not distinguish these three layers on the FPGA side. So this header is processed parallel to the tasks and thereby verified w.r.t. the checksum. After that, the time-to-life field is adjusted and a new checksum is computed, in order to assemble a package containing the result of the computation which is resent to the FPGA Worker Driver.

This framework takes the possibility of errors into account and implements recovery routines: On one node, the CPU periodically sends *Hello* packets to all FPGAs. If there is no answer until a timeout occurs, the FPGA is marked as failed. This issues two actions:

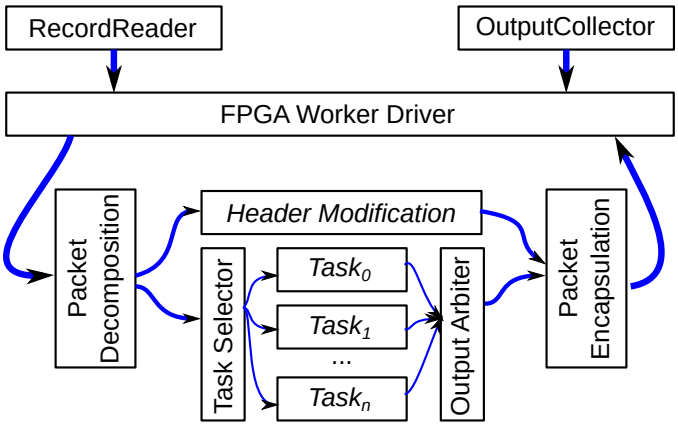

Fig. 7. Dataflow in the scalable MapReduce framework of Yin et al.

1. The FPGA is re-flashed with the initial configuration and
2. all tuples that were processed on the FPGA when the failure was detected are resent again.

In order to resend the tuples correctly, the FPGA Worker Driver comes with an additional backup FIFO, where the most-common data is put into and retreived from if necessary.

Experiments For experimental evaluation, Yin et al. had a standalone-setup, i.e. one CPU worker and up to three FPGAs with different number of concurrently running tasks. They evaluated first matrix multiplication and consecutively the computation of PageRank with different input sizes, results are summed up in figure 8 and table 1.

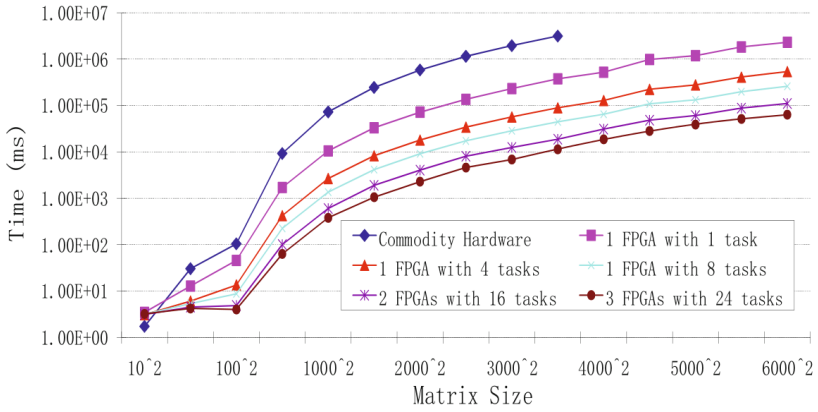

Fig. 8. Results of matrix multiplication with Yin et al.'s framework.

Matrix multiplication of matrices $A = (a_{ij})$ and $B = (b_{jk})$ seems to be implemented by computing the inner product of rows of A and columns of B, i.e. $\sum_j a_{ij} \cdot b_{jk}$ in the map phase, and reassembling the matrix in the reduce phase. The blue diamonds in figure 8 show the performance of this job run on a CPU as reference.

Number of Nodes/Edges	Commodity Hardware	Scalable framework with FPGA
102448/2162410	5.356s	2.056s
916417/6078254	38.211s	9.702s
1211756/9076553	91.145s	24.235s

Table 1. Results of PageRank computation with the scalable MapReduce framework.

Expectedly, for small (10×10) matrices the jobs run fastest on CPU only, as there is no network communication overhead. The first performance increase shows up, when the number of tasks which run concurrently on the FPGA gets from 1 to 4 and to 8. The solution on the external chip is already about 15 to 20 times faster than the reference time.

When using three FPGAs with a total of 24 tasks, the performance gains again, but not linearly since the bandwith of the network could not handle more.

4.2 Axel

In this subsection we take a look at Axel, a heterogeneous cluster built by Tsoi and Luk [TL10] which consists of nonuniform nodes. This means, every node in the cluster is equipped with different processing elements wich CPU, GPGUs and FPGAs. We will focus on the integration of FPGAs into Axel and how their benefits are exploited.

Architecture On a cluster level, Axel follows the "nonuniform nodes uniform system" approach, which means, on a system's perspective, the nodes are all equally equipped, but a node itself consists of more than one type of processing element. In this case, the processing elements are a CPU with four cores, a GPGU and an FPGA all interconnected via PCI-Express. The nodes then are interconnected via ethernet.

The nodes in Axel run Linux, a set of tools specific to the processing elements and a custom resource management system, which is used to allocate GPU and FPGA. This is necessary since computation tasks on GPU and FPGA are not managed by the operating system and so context switches between tasks are not supported.

Developing a MapReduce application Tsoi and Luk focus on explaining a workflow for utilizing FPGAs and we can learn a lot from this, so we also include this.

The user has to define input and output types, as well as a function which reads input and the functions map and reduce [YTT+08]. Axel is not equipped with a distributed file system or database, so the user has to manually partition data to the different nodes in the cluster. For utilizing the diverse processing elements, the user can choose which part of the computation is executed where; e.g. the FPGA takes care of the map function and the CPU of the reduce function.

In order to estimate the efficiency of this distribution of input data and computation, one can make use of the so called Hardware Abstraction Model. This model states for each processing element the available computation capabilities, local memory and communication devices. If the estimated performance is not good enough, the distribution of the input data or the worker processes has to be adjusted and tested again. Otherwise all modules are compiled according to their type of processing element and a top level application has to be written.

Experiments The experimental evaluation of Axel was done with the so called n-body simulation which models the interaction between particles. The computation task is described in algorithm 1. When analyzing this algorithm, the inner j-loop can be identified as hot spot and will be computed by one map function. The whole i-loop is computed in the map phase and finally the k-loop in the reduce phase. In order to minimize communication overhead, they choose to let the CPU handle the reduce phase, and assign the map tasks to GPGU and FPGA.

Algorithm 1 N-body force computation as presented in [TL10]

1: **for** $i := 1..N$ **do**
2: **for** $j := 1..N$ **do**
3: $r.x = p[i].x - p[j].x$
4: $r.y = p[i].y - p[j].y$
5: $r.z = p[i].z - p[j].z$
6: $d = (r.x^2 + r.y^2 + r.z^2 + \varepsilon)^{-1.5}$
7: $s = p[j].m \cdot d$
8: $a[i].x+ = r.x \cdot s$
9: $a[i].y+ = r.y \cdot s$
10: $a[i].z+ = r.z \cdot s$
11: **end for**
12: **end for**
13: **for** $k := 1..N$ **do**
14: $p[k].x = p[k].x + p[k].vx$
15: $p[k].y = p[k].y + p[k].vy$
16: $p[k].z = p[k].z + p[k].vz$
17: $p[k].vx = p[k].vx + a[k].x$
18: $p[k].vy = p[k].vy + a[k].y$
19: $p[k].vz = p[k].vz + a[k].z$
20: **end for**

With their hardware model, Tsoi and Luc can calculate an upper bound for the compution time. With x being the number of multipliers, y being the number of adders and f being the frequency of the FPGA, the time for the $N \cdot (N-1)$ executions of all 17 operations can be estimated with the following equation:

$$T_{cf} = N \cdot (N-1) \cdot \frac{max\{\frac{8}{x}, \frac{9}{y}\}}{f} \tag{1}$$

For the FPGA they used, this equation yields $\approx N \cdot (N-1) \cdot 3ns$. Yet, this model does not take into account other resources on FPGA chips as discussed in section 1. Also the implementation of the circuit does not directly reflect the sequential algorithm but yields a deeply pipelined data path which is 132 clock cycles long. For one P_i, all P_j values are put into the pipeline, one per clock cycle. After this, the generated A value is returned, the pipeline is emptied and the next P_i is computed.

With this coding scheme, the expected performance of computing the simulation with the Dubinski data set (a benchmark with 81920 particles) was 20.21s. But when they did performance measurements, the algorithm took about 47 seconds to complete. It turned out that the difference comes from memory read overhead as the model does not take into account the method of operation of the memory controller and the behaviour of the used DDR2 memory.

Since the resource utilization was below 10% of the chip, they decided to duplicate the pipeline logic and thereby effectively put ten cores onto the FPGA which compute ten iterations of the outer P_i loop in parallel. With this optimization, the

execution time of the algorithm was only 5.62 seconds and including file i/o and initial configuration time of the FPGA 8.43 seconds.

Results We will now have a look into the measurements of this algorithm run on Axel's CPUs, GPGUs and FPGAs which are summed up in table 2. The first measurement as a reference was the single-threaded execution of this algorithm on the host CPU (AMD Phenom Quad-Core) which can directly be compared to the multi-threaded execution on the same CPU. The latter was a little less than four times faster, which can be expected as there is some communication overhead on shared memory access.

Processing element	Execution time	Total time
CPU (single threaded)	99.3s	99.5s
CPU (multi threaded)	29.1s	29.3s
GPU	9.26s	9.53s
FPGA (one core)	46.6s	48.9s
FPGA (ten cores)	5.62s	8.43s

Table 2. Measurements of N-body simulation in Axel

The benchmark for the nVidia Tesla C1060 GPU was to compute only the acceleration of particles, not the position updates which were still due to the CPU. With one thread per P_i, the graphics card performed three times faster than the CPU. Here it is worth mentioning that the results differ since the floating point computations are implemented differently.

Finally, we see the FPGA performing moderately if configured to have one core. But if it has ten cores, i.e. nearly the complete chip is used, it actually beats the performance of the graphics card. This might be surprising if one only considers the clock rate, which is 400MHz for the FPGA, 1.2GHz for the GPU and 2.3GHz for the CPU. But the possibility of creating a deeply pipelined circuit specific to this particular problem renders this performance possible.

4.3 FPMR

In 2010, Shan et al. presented FPMR, a MapReduce framework for FPGAs [SWY+10]. In order to use their framework, one has to write mapper and reducer modules. FMPR then takes care of placing these modules on a chip according to configurable numbers of map and reduce tasks as well as space limitations of the FPGA. Moreover, FPMR places modules needed for task and data management on the chip, such that this approach is FPGA-only.

Architecture In order to understand the architecture of FPMR, we look at first at the general data flow from input to output data. Then we examine the task management structures that take care of scheduling.

The **data path** FPMR assumes that input data already consists of <key, value>-pairs. So the host has to take care of preparing the input data accordingly. The prepared input tuples are sent over a system bus like PCI-E to the FPGA. The so called *data controller* stores tuples into a local memory (e.g. SDRAM). The mappers get tuple by tuple from this memory and process them. The resulting intermediate tuples are then stored in an *on-chip memory*, i.e. a part of the FPGA is used to implement a RAM. These intermediate tuples go to the according reducer

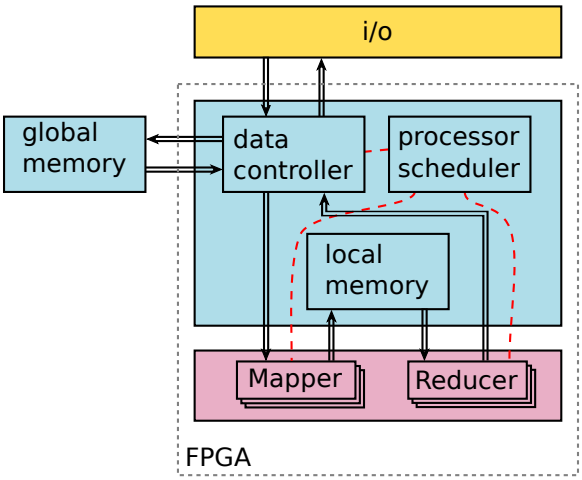

Fig. 9. Dataflow in FPMR

and the generated result data is stored via the data controller in local memory again. After completion the data is sent back to the host system.

For the **control path** the process scheduler maintains four queues: one holding the ids of idle mappers and one holding pending map tasks and analogously two queues for the reducers. It actively schedules pending tasks to available processors. When a processor finishes its task, its id is put back into the idle-mappers queue. The claim of Shan et al. is, that "in such a scheme, mappers and reducers cooperate with each other to keep all the processors as busy as possible". This shows a big difference to the control idea of Hadoop. In Hadoop, map and reduce are two clearly seperated phases.

Experiments with RankBoost For evaluating the performance of their framework, the RankBoost algorithm was used. This algorithm classifies documents $d = 0..N_d$ according to up to N_f query features, which are stored in a feature vector $f_i(d)$. The ranking is done with respect to the so called distribution $D(d_k, d_l)$ which is positive, if d_k is more important than d_l. In order to reduce compuational complexity they introduced a value π for every document which is computed as $\sum_{d'}(D(d', d) - D(d, d'))$.

The RankBoost algorithm iteratively computes the result in a given number T of rounds. In every round, the WeakLearn procedure given in algorithm 2 is executed and with the returned values of h_t and α_t the distribution weights for all pairs (d_k, d_l) are updated. The bin_f values we see in the description of WeakLearn come from discretizing the continuous components of the feature vectors.

As they WeakLearn can be figured out to be the hot spot in this computation [XCG+09], we here focus on its implementation with FPMR and its execution on the FPGA. The procedure is split up such that one map function computes the body of the loop in line 3, and the reducer aggregates the values as in line 6. Even the updates of h_t and α_t from line 10 to the end are done on the CPU. As the reducer's job is to compute the sum over all values and they decided to start one job per feature vector (i.e. one job per outer loop of the algorithm), just one reducer is needed and the remaining space on the chip can be used for mappers.

In the FPMR implementation, one mapper reads the values of bin_f and π for the input d. While $\pi(d)$ is needed as the input for a floating point adder, $bin_f(d)$ serves as address for reading and writing to a local RAM module, which stores $hist_k$. This way, the reducers here have a certain state which is used if two documents are in

Algorithm 2 WeakLearn procedure used in RankBoost [SWY+10]

Input: $\pi(d)$ for current round, $bin(d)$ for all documents
Output: Weak hypothesis h_t and weight α_t

for $k := 0..N_f$ **do**
  for $d := 0..N_d$ **do**
    $hist_k(bin_f(d)) := hist_k(bin_f(d)) + \pi(d)$
  end for
  for $i := N_{bin} - 1..0$ **do**
    $integral_k(i) := hist_k(i) + integral_k(i+1)$
  end for
end for
$integral_{fmax} := \max\{integral_{fmax}(bin_{max})\}$
for $d := 0..N_d$ **do**
  if $bin_{fmax}(d) > bin_{max}$ **then**
    $h_t(d) := 1$
  else
    $h_t(d) := 0$
  end if
end for
$\alpha_t := \frac{1}{2} \ln\left(\frac{1+integral_{fmax}}{1-integral_{fmax}}\right)$

the same bin. The reducer then sequentially builds the sum over all bins and stores in every step the current maximum integral value.

Results For evaluating the FPMR implementation of WeakLearn in RankBoost, Shan et al. used an Intel Pentium 4 Processor with 3.2GHz with 4GB DDR2 memory as host system and as FPGA the Altera Stratix II EP2S180F1508 running at 125MHz. As benchmark data "a real world dataset for a commercial search engine" was used. They compared the CPU-only implementation with the use of a varying amount of mappers and the use of the common data path, which is shown in table 3

	Without CDP				With CDP			
#mappers	WL [s]	Total [s]	S(WL)	S(Total)	WL [s]	Total [s]	S(WL)	S(Total)
1	320.89	321.96	0.325	0.327	320.9	321.96	0.33	0.33
2	160.89	161.52	0.65	0.652	160.5	161.52	0.65	0.65
4	80.224	81.293	1.3	1.296	80.22	81.293	1.3	1.30
8	40.112	41.181	2.6	2.559	40.11	41.181	2.6	2.56
16	20.056	21.125	5.2	4.988	20.06	21.125	5.2	4.99
32	10.09	11.159	10.33	9.443	10.09	11.159	10.33	9.44
52	6.228	7.297	16.74	14.44	6.228	7.297	16.74	14.44
64	6.228	7.297	16.74	14.44	5.107	6.7	20.42	17.06
128			–		2.616	3.685	39.87	28.59
146			–		2.242	3.311	46.52	31.82
Reference	104.3	105.37	1	1	104.3	105.37	1	1

Table 3. Results of RankBoost computation with FPMR, S denotes the speedup compared to the reference implementation

As seen in this table, without using the common datapath, the performance can be massively increased but the gain stops at 52 mappers. This is, because in one clock cycle only 16 bin_f and 1 π value can be retreived from the DDR2-memory and it takes 52 cycles for one mapper to complete these data. So, 52 mapper can run in parallel without being idle, but adding more mappers does not work because

of the performance bottleneck brought by the need of reading π. Since the same π is read by multiple mappers, a solution is, to put π in the common data path and read the same values only once, even prefetch them. This way, the space-limited maximum number of 146 mappers could be placed on the Stratix chip. With these figures, we see that not only the weak learn procedure is in fact the hot spot of the RankBoost computation, but also the performance boost of up to a factor of 32.

5 Comparison

In this section, we compare the three systems and their perception of MapReduce.

5.1 Architecture

The first framework bases on Hadoop's architecture. It hooks into the implementation of mappers and reducers and forwards their computation tasks to the FPGA. Axel is a whole cluster where every node has several processing elements, including an FPGA. Here the user can decide which task is done by the FPGA and which are done by CPU or GPGU. FPMR is a solution that is placed just on one FPGA and is driven by a normal CPU. All computation is done on the FPGA, the user is limited by the chip size.

This short comparison already shows big differences: FPMR is not built with the notion of a cluster in mind, so there is no distributed data involved. Also the data—especially the intermediate tuples—never leave the FPGA (except for its local RAM). So this system seems best fitted for computations that narrow down the amount of data, like the RankBoost example. On the other hand, Axel is an architecture based on a cluster setup. Here the user has the freedom to put computations to the FPGA that fit the size and computation power of the FPGA and put other computations to other processing elements. Still the user has to take care of interaction between the several program parts and has to manually distribute the data over the cluster. The last point is no necessity for users of the first framework, since data distribution is there done by Hadoop. Also this seems to be the only framework that can be used with existing infrastructure. One only has to plug in an FPGA board in the Hadoop cluster nodes and adjust of the jobs accordingly.

In terms of the approaches we discussed in section 3, FPMR is an all-FPGA solution, while the other two follow the shared-management way. We did not see a FPGA-slave implementation and this is understandable by looking at the MapReduce framework of Yin et al. The management components that they placed on the chip are due to the scheduling of tasks. Without this local scheduling the benefit of extremely fast parallel computation on the FPGA would not be possible.

5.2 Understanding of MapReduce

We also see, that the term "MapReduce" is understood differently by these three systems. FPMR claims to be MapReduce because of the computational model which consists of a map and a reduce function. Whereas Axel and the framework by Yin et al. have not only the computational model but also the architectural model, where the execution of the map function is moved towards the data such that communication is minimized.

But all three do not provide the ease of a MapReduce framework where well defined interfaces seal off the technical details from the user. In order to get the full benefits of the hardware accelerators, one has to either leave the functional aspect from the map and reduce functions behind and introduce a state to the mappers as in the RankBoost example. Or sophisticated sharing of values like in the Common Data Path option of FPMR is done.

5.3 Limitations

All approaches we have seen rely on the fact that the data has a fixed format, e.g. integer or IEEE double precision. If the performance gain of using FPGAs is still as good for data with arbitrary length like Strings remains open.

Existing MapReduce jobs cannot easily be translated on either of the FPGA solutions. On one hand, the high-level synthesis does not necessarily produce good hardware architectures as already stated. On the other hand, many aspects of the algorithm may not be feasible for FPGA computation, e.g. the use of random-access or pointer-based data structures or the way how data is sequentially accessed [HVG+07]. So the present algorithms may have to be rewritten to fit to the FPGA.

As discussed before, FPMR can only be used for jobs that produce an amount of intermediate data small enough to fit on the FPGA. Other jobs that produce as many intermediate keys (like matrix multiplication) can only be executed if the input size is small enough.

6 Conclusion

We have seen that by using FPGAs, the computation times of MapReduce tasks can be shrinked down by an order of magnitude. But still this evolution is only at the beginning. The framwork by Yin et al. and FPMR give a glimpse of how frameworks can look like, but they either rely on certain FPGA hardware, or on a certain shape of MapReduce job. Axel tells us, how a specialized cluster can exploit acceleration hardware in a sensible way. But all three show how FPGAs can leverage the inherent parallelism of MapReduce jobs and besides GPUs.

However, a last and important point is usability. While MapReduce developers nowadays have big well-documented frameworks and can use full-fledged development environments in order to program jobs, the introduction of FPGAs brings new obstacles into their world: As FPGAs are programmed with Hardware-description language and high-level synthesis is not far enough, either the software developers have to learn how to program hardware, or hardware developers have to be hired and become acquainted with the big-data domain.

So finally there is more research to be done, before FPGAs can advantageously be used for our everyday MapReduce operations.

References

[Alt14] Altera. About stratix family high-end fpgas and socs, July 2014. URL: `http://www.altera.com/devices/fpga/stratix-fpgas/about/stx-about.html`.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[Ges14] Ralf Gessler. *Entwicklung Eingebetteter Systeme.* Springer Fachmedien Wiesbaden, Wiesbaden, 2014.

[Had14] Apache Hadoop. The apache hadoop website, July 2014. URL: `http://hadoop.apache.org/`.

[HVG+07] Martin C Herbordt, Tom Vancourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug Disabello. Achieving High Performance with FPGA-Based Computing. *Computer*, 40(3):50–57, March 2007.

[Int14] Intel. Intel® core™ i7-4790k processor – specifications, July 2014. URL: `http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz`.

[MD14] Pedro Martins Dusso. Optimizing sort in hadoop using replacement selection. Master's thesis, University of Kaiserslautern, 5 2014. URL: `fileadmin/publications/2014/MScThesisDusso2014.pdf`.

[MT09] Rene Mueller and Jens Teubner. Fpga: What's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1559845.1559965`.

[MT10] Rene Mueller and Jens Teubner. Fpgas: A new point in the database design space. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 721–723, New York, NY, USA, 2010. ACM.

[SJD13] Johannes Schildgen, Thomas Jörg, and Stefan Deßloch. Inkrementelle neuberechnungen in mapreduce. *Datenbank-Spektrum*, 13(1):33–43, 2013. URL: `http://dx.doi.org/10.1007/s13222-012-0109-3`.

[SWY+10] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR: MapReduce Framework on FPGA - A Case Study of RankBoost Acceleration. pages 93–102, 2010.

[TL10] Kuen Hung Tsoi and Wayne Luk. Axel: A Heterogeneous Cluster with FPGAs and GPUs. pages 115–124, 2010.

[Weh14] Norbert Wehn. Implementation Styles, 2014.

[XCG+09] Ning-Yi Xu, Xiong-Fei Cai, Rui Gao, Lei Zhang, and Feng-Hsiung Hsu. Fpga acceleration of rankboost in web search engines. *ACM Trans. Reconfigurable Technol. Syst.*, 1(4):19:1–19:19, January 2009.

[Xil14] Inc Xilinx. 7 Series FPGAs Overview, 2014. URL: `http://www.xilinx.com/support/documentation/data\_sheets/ds180\_7Series\_Overview.pdf`.

[YLH12] Dong Yin, Ge Li, and Ke-di Huang. LNCS 7469 - Scalable MapReduce Framework on FPGA Accelerated Commodity Hardware. pages 280–294, 2012.

[YTT+08] Jackson H.C. Yeung, C.C. Tsang, K.H. Tsoi, Bill S.H. Kwan, Chris C.C. Cheung, Anthony P.C. Chan, and Philip H.W. Leong. Map-reduce as a Programming Model for Custom Computing Machines. *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 149–159, April 2008.