# Accelerating the Cube Cut Problem with an FPGA-augmented Compute Cluster

**Tobias Schumacher, Enno Lübbers, Paul Kaufmann, and Marco Platzner**

Paderborn Center for Parallel Computing (PC$^2$)
University of Paderborn
*E-mail:* {*tobe, enno.luebbers, paul.kaufmann, platzner*}*@uni-paderborn.de*

The cube cut problem refers to determining the minimum number of hyperplanes that slice all edges of the $d$-dimensional hypercube. While these numbers are known for $d \leq 6$, the exact numbers for $d > 6$ are as yet undetermined. The cube cut algorithm is able to compute these numbers, but is computationally extremely demanding. We accelerate the most time-consuming part of the cube cut algorithm using a modern compute cluster equipped with FPGAs. One hybrid CPU/FPGA node achieves a speedup of 27 over a CPU-only node; a 4-node hybrid cluster accelerates the problem by a factor of 105 over a single CPU. Moreover, our accelerator is developed using a structured, model-based approach for dimensioning the hardware implementation and analyzing different design points.

## 1 Introduction

The $d$-dimensional hypercube consists of $2^d$ nodes connected by $d \times 2^{d-1}$ edges. A prominent and still unsolved problem in Geometry is to determine $C(d)$, the minimal number of hyperplanes that slice all edges of the $d$-dimensional hypercube. An upper bound for $C(d)$ is given by $d$. This is achieved, for example, by the $d$ hyperplanes through the origin and normal to the unit vectors. For $d \leq 4$, it has been known that at minimum these $d$ hyperplanes are required. For $d = 5$, it was shown only a few years ago that actually five hyperplanes are required[1]. Surprisingly, several sets of only 5 hyperplanes have been found that slice all edges of the 6-dimensional hypercube.

The cube cut problem relates to the question of linear separability of vertex sets in a $d$-hypercube. Linear separability plays also a central role in the areas of threshold logic[2], integer linear programming[3] and perceptron learning[4]. In threshold logic, a threshold function $f$ of $d$ binary variables is defined as $f(x_1, x_2, \ldots, x_d) = \{1 : \text{iff } \sum_{i=1}^{d} w_i x_i \geq T; 0 : \text{iff } \sum_{i=1}^{d} w_i x_i < T\}$, with $w_i, T \in \mathbb{R}$. The input variables span an $d$-hypercube and $f$ separates input variables (vertices) mapped to 0 from vertices mapped to 1 by a linear cut through the $d$-hypercube. Here, for example, it has been shown that $C(d)$ is the lower bound for the size of a threshold circuit for the parity function[2].

In the past, much work was done for finding $C(d)$[5–7]. On one hand, scientists approach this problem analytically and try to find a mathematical proof for this number. While an analytical solution would be extremely desirable, such a proof seems to be very hard to obtain. On the other hand, the problem can be approached computationally by evaluating *all possible slices* of the $d$-dimensional hypercube using *exhaustive search*. While the computational method lacks the mathematical strength of a formal proof, it has indeed brought remarkable success during the last years[1,8].

The complexity of computing all possible slices for higher dimensions leads to very long runtimes. The main contribution of this paper is the acceleration of the most runtime-intense part of the cube cut algorithm by a high-performance compute cluster utilizing FPGAs. A second contribution is the use of an architecture and application model that helps us to structure the implementation of the FPGA accelerator for maximum efficiency. Section 2 gives an overview of the cube cut problem and the algorithm used to solve it. In Section 3, we introduce the basics of the FPGA accelerator modeling. The implementation decisions we have taken based on information gathered from the model are detailed step-by-step in Section 4. This section also presents performance data and compares them to a software-only implementation on the same compute cluster. Finally, Section 5 concludes the paper and presents an outlook into future work.

## 2 The Cube Cut Algorithm

The main idea of the computational solution to the cube cut problem is to generate all possible hyperplanes that slice some of the edges of the $d$-dimensional hypercube. Because the $d$-dimensional hypercube contains $n = d \times 2^{d-1}$ edges, such a cut can be described by a string of $n$ bits. Each bit position in that string corresponds to a specific edge and is set to 0 when the edge is not sliced by a cut and to 1 when the edge is sliced, respectively. For example, for $d = 2$ a cut is described by a string of 4 bits where $(\{1010\}, \{0101\})$ forms a minimal set of hyperplanes cutting all edges.

The cube cut algorithm consists of three phases. First, all possible cuts for a specific $d$ are generated. This results in a huge list of bit strings. Second, this list is reduced by a technique described below. Third, the remaining set of bit strings is searched for a subset of bit strings that represents a set of cuts slicing all edges of the $d$-dimensional hypercube. This is achieved by identifying a minimum set of bit strings $b_0, b_1, \ldots, b_{C(d)-1}$ where each of the $n$ bits is set in at least one of the strings. Since the number of possible cuts grows rapidly with $d$, searching the full set of bit strings for a minimum number of cuts is practically impossible. Hence the reduction of the list of bit strings in phase two is of utmost importance to reduce the complexity for phase three. The reduction technique is based on the property of dominance.

Consider a cut $c_1$ that slices the edges $E = \{e_0, e_1, \ldots, e_{k-1}\}$ and another cut $c_2$ that slices all edges in $E$ plus some additional ones. Obviously, cut $c_1$ can be discarded since we know that another cut exists that already slices all edges in $E$. We say that $c_2$ *dominates* $c_1$. Formally:

$$b \text{ dominates } a \Leftrightarrow \forall i : (\neg a_i \vee b_i) = 1; i = 0, ..., n-1 \quad (1)$$

From the complete set of bit strings we can typically discard a large number of cuts that are being dominated. Rather than checking all pairs of bit strings for dominance, we employ the following algorithm: First, we generate two separate lists $A$ and $B$ from the initial list of possible cuts. $B$ is initialized with all bit strings that slice a maximal number of edges. Note that these bit strings can not be dominated. Assume that every element of $B$ contains exactly $k_{max}$ ones. $A$ is initialized with all bit strings containing exactly $k_{max} - 1$ ones. Every element in $A$ is then compared to the elements in $B$. The elements of $A$ that are dominated by an element of $B$ are discarded. After this step, the remaining elements of $A$ are known to be non-dominated by any of the remaining elements of the initial list as the latter contain fewer ones and therefore slice fewer edges. The remaining elements of $A$ are thus added to the set $B$. Then, a new set $A$ is created containing all elements with

$k_{max} - 2$ ones and checked against $B$, and so on. The algorithm runs until all elements of the original list have been checked for dominance. The result of this algorithm is a reduced list of bit strings where all irrelevant cuts have been discarded.

Although phase two of the cube cut algorithm consists of rather simple bit operations on bit strings, it shows a very high runtime in software. This is mainly due to the fact that the required bit operations and the lengths of the bit strings do not match the instructions and operand widths of commodity CPUs. FPGAs on the other hand can be configured for directly operating on a complete bit string at once and hence exploit the high potential of fine-grained parallelism. Moreover, provided sufficient hardware area is available, many of these operations can be done in parallel with rather small overhead. An FPGA implementation can therefore also leverage both the fine- and coarse-grained parallelisms inherent in phase two of the cube cut algorithm. We have implemented that phase of the cube cut algorithm in hardware and mapped it to an FPGA (see Section 4).

## 3 Modeling FPGA Accelerators

Many common modeling and analysis techniques for parallel algorithms, e.g., PRAM, describe algorithms by simple operations and express the execution times by counting the number of such operations performed. While these models are of great importance from a theoretical point of view, their usefulness is often also rather limited when it comes to implementations on real parallel machines. The major problem with these models is that they consider only the time spent for performing calculations and do not regard the time spent for data accesses, i.e., memory and I/O.

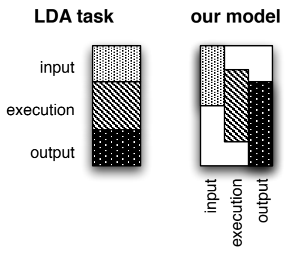


Figure 1. Modification of the LDA model

To overcome this limitation, the LDA (latency-of-data-accesses) model[9] was created previously. The LDA technique actually subsumes an architecture model, an execution model, and an execution time analysis. The architecture model defines the various properties of the target architecture, especially the available memory hierarchies together with their latency and bandwidth parameters. The execution model describes algorithms in terms of LDA operations. LDA operations are classes of original machine instructions that can be characterized by common latencies. For memory accesses, the model differentiates between accesses to L1-/L2-cache, main memory, or several levels of remote memory. *Tasks* are the basic units of computation. A task consists of an *input* phase (synchronize with and receive inputs from preceding tasks), an *execution* phase (computation and memory access), and an *output* phase (synchronize with and send output to subsequent tasks). The three phases are executed in sequence as shown in Figure 1. Control flow is modeled by connecting several tasks. The LDA model is not cycle-accurate but aims at providing a means to analyze different application mappings on parallel architectures and to estimate their performance either analytically or by simulation. Simulators have been developed atop the LDA model and used successfully to investigate applications on SMP machines[10].

While the LDA model takes communication into account, it is not well-suited for modeling FPGA accelerators. This is for a number of reasons. First, FPGA cores can utilize the unique capabilities of programmable hardware in many different ways. Besides exploiting huge amounts of fine-grained parallelism at the bit-level, custom cores can also employ SIMD style parallelism, pipelining and systolic execution modes. The execution phase of such a core is not at all sequential, as the LDA model assumes. Especially when the execution phase is interleaved with load and store phases, the LDA model with its sequential input–execute–output semantics is not applicable. However, one feature of the LDA model that we think transfers well to FPGA accelerators is its support of data accesses with different latencies. This is relevant to FPGA designs as data accesses can target anything from distributed RAM over BlockRAM and off-chip SRAM and DRAM to memories connected over I/O busses.

We therefore work on an LDA-based modeling technique for FPGA accelerators. While we retain LDA's three-phase structure, we allow for overlapping phases and consider the fine-grained parallelism of hardware. Figure 1 shows the main difference between our model and LDA. In our model, the input, execution, and output phases may occur in parallel with arbitrary time offsets. All phases are characterized by their *bandwidth*, while the architecture nodes implemented on the FPGA also possess an *area* parameter. While the bandwidth of a communication phase is dependent on architecture parameters like bus width or clock frequency, the bandwidth of an execution phase depends on a larger set of design parameters. For example, the designer can vary pipeline depth, pipeline width, and the delay of one pipeline stage, all of which have direct impact on the area requirements, the attainable clock frequency and the required I/O bandwidth.

In the work presented in this paper, we use the LDA-based model to describe and reason about different design options for an FPGA core accelerating the cube cut algorithm.

## 4 Implementation and Results

### 4.1 Target platform

The implementation of our cube cut algorithm is targeting the Arminius compute cluster provided by the Paderborn Center for Parallel Computing[11]. This cluster consists of 200 compute nodes, each one equipped with two Xeon processors running at $3.2$ GHz and using $4$ GB of main memory. The nodes are connected by a $1$ Gbit/s Ethernet and an additional $10$ Gbit/s Infiniband network for high speed communication.

Additionally, four nodes of the cluster are equipped with an AlphaData ADM-XP FPGA board[12]. The user FPGA of this board, a Xilinx XC2VP70-5, is connected to the host system by a PCI 64/66 bridge implemented in an additional Virtex-II FPGA. This bridge provides a rather simple 64bit wide multiplexed address/databus to the user FPGA, called `localbus`. The user FPGA can be clocked from 33 to $80$ MHz, and data can be transfered using direct slave cycles or by DMA transfers. Two DMA channels are available which can operate in different modes. For our application, the most useful DMA mode is the demand mode which is optimized for connecting to a FIFO. In this mode, the DMA channels block the host application running on the CPU until the FIFOs are ready for data transfers. By experimenting with micro benchmarks we were able to determine a bidirectional bandwidth of $230$ MB/s using demand mode DMA.

A sketch of the architecture model for our node setup is shown in Figure 2. The algorithm kernel that is to be accelerated on the FPGA is logically located between the two

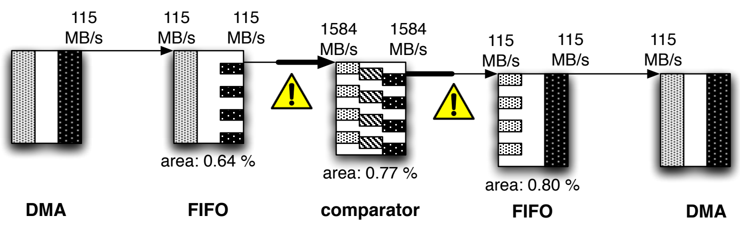


Figure 2. Architecture model

FIFOs (one for input, one for output). These FIFOs in turn connect to the DMA controller and are also used to convert between bit widths and clock frequencies that might be different on the PCI bus and the user FPGA. Figure 2 shows an example where the DMA channels transmitting data over the PCI bus form a bottleneck. In our model, the DMA channels and the FIFOs are responsible for data transfers and do not have any internal execution phase.

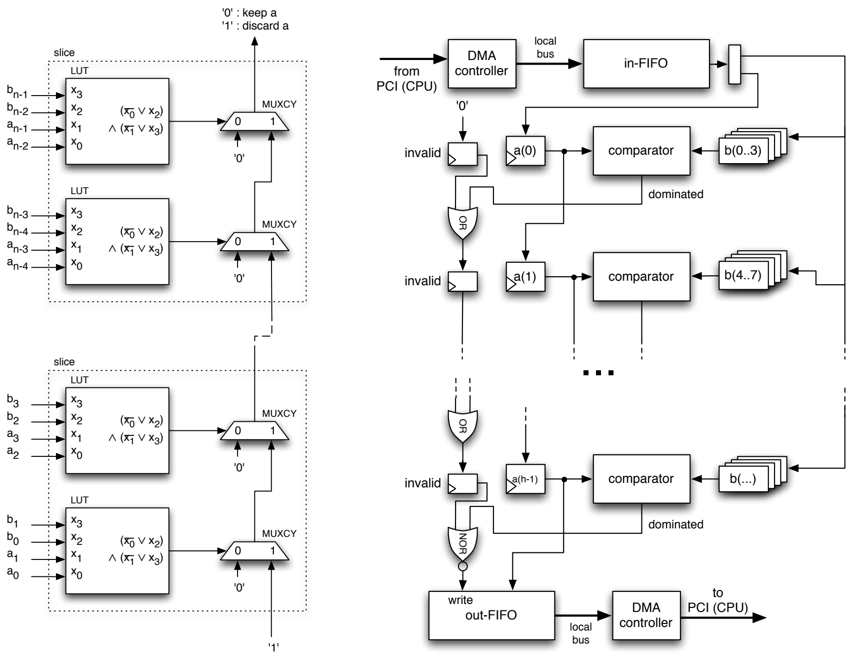
## 4.2 FPGA core for checking dominance

The central element of the algorithm outlined in Section 2 is the comparison of two bit strings for dominance. A dominance comparator takes as inputs two $n$-bit strings $a$ and $b$ from lists $A$ and $B$, respectively. It checks every bit of $a$ against the corresponding bit of $b$ to detect a possible dominance, according to Equation 1. We rely on an optimized design that makes efficient use of the LUTs and the carry logic offered by FPGAs. The design mapped to Xilinx Virtex-II Pro technology in shown in Figure 3(a). Every 4-input LUT checks two bits of the input strings $a$ and $b$ for dominance, e.g., $dom = (\neg a_0 \vee b_0) \wedge (\neg a_1 \vee b_1)$. As a result, $dom$ is one when this part of $b$ dominates the appropriate part of $a$. All the LUTs required for a comparator of given bit string length $n$ are connected to a $n$-input AND gate formed by the slice's carry logic, driven by a multiplexer (MUXCY). The output of this multiplexer $c_{out}$ is tied to constant zero when $dom = 0$, and to the value of $c_{in}$ else. The first $c_{in}$ of this chain is tied to a constant one; the last $c_{out}$ is the result of the complete dominance check. This approach follows the design pattern for wide logic gates[13] and modifies it to include the dominance check.

In the following, we focus on an FPGA accelerator for $d = 6$ which results in bit strings of length $n = d \times 2^{d-1} = 192$ bit. On our target device, an XC2VP70-5, a comparator for this length consumes 0.77 % of the FPGA's logic resources and can be clocked at a maximum of 66 MHz. Using the design of Figure 3(a), one pair of bit strings can be checked for dominance in just one clock cycle. Under the assumption that an element from list $B$ has already been loaded into the FPGA, the comparator design needs to read one element from list $A$ per cycle. In the worst case no $a \in A$ is dominated by any of the elements of $B$. Then, the comparator will also output one element per cycle. The required combined input and output bandwidth of the single comparator for $n = 192$ is therefore:

$$bandwidth_{req} = 192\,\text{bit} \times 66\,\text{MHz} \times 2 = 3168\,\frac{\text{MB}}{\text{s}}$$

However, the available bandwidth for the input and output channels of the DMA controller is only 230 MB/s – roughly $\frac{1}{14}$ of the combined data bandwidth requested by the



(a) Efficient comparator implementation

(b) Comparator pipeline

Figure 3. Dominance-check core architecture

comparator design. Such an FPGA accelerator would be perform poorly as the DMA bandwidth over the PCI bus forms a severe performance bottleneck. This situation is reflected by the model shown in Figure 2. Moreover, the single comparator design utilizes only 0.77 % of the available FPGA area which is rather unsatisfying.

In order to reduce the comparator's bandwidth requirements until it matches the DMA channels' bandwidth, we modified the design such that $m$ elements of the list $B$ are stored on the FPGA. When the element $a$ is not dominated by any of the stored elements $b$, the comparator will need a new input value and write a new output value only every $m$-th cycle. To increase the FPGA utilization, we have two options. First, we could instantiate $l$ comparator circuits onto the FPGA that operate in parallel. This would, however, again increase the necessary bandwidth by the factor $l$. The second option is to arrange $p$ comparators in a pipelining fashion. Each stage of the pipeline stores $m$ elements of the list $B$ and compares them with an element $a$. Then, the element $a$ proceeds to the next pipeline stage where it is compared to the next bunch of $m$ elements of $B$. As soon as an element $a$ is found to be dominated, it is invalidated. While such an element continues to run through the pipeline as a "bubble" for the reason of keeping all pipeline stages synchronized, the output FIFO will not accept this element. The resulting design for $m = 4$ is shown in Figure 3(b).

We easily conclude that given the 0.77 % area utilization of one comparator, we could fit 130 comparators on the FPGA. Considering that we have to assign $m = 16$ elements of $B$ to each comparator stage to reduce the bandwidth to $\frac{192\ \mathrm{bit} \times 66\ \mathrm{MHz} \times 2}{16} = 198\ \frac{\mathrm{MB}}{\mathrm{s}}$, we can still map 100 comparator stages to the target device. As a result, every element $a$ sent to the FPGA is compared to at most 1600 elements $b$. At a clock rate of 66 $\mathrm{MHz}$ and assuming that the pipeline is completely filled with valid elements $a$, we achieve a performance of roughly $10^{11}$ bit string comparisons per second.

## 4.3 The host application

The software part of the cube cut algorithm, phase two, consists of feeding the FPGA with proper lists $A$ and $B$ and waiting for results to be read back. The host application consists of two threads, a sender and a receiver, that perform these tasks. The sender loads the $b$'s into the comparators' memories and streams the $a$'s through the pipeline, while the receiver simply waits for bit strings that were not dominated and writes the values back into a buffer. The buffers for $a$'s and the results are then swapped, new $b$'s are loaded, and the threads proceed as before.

We have also parallelized the host application by distributing the list $A$ over different compute nodes such that only a fraction of the overall comparisons needs to be performed on one node. Since the Arminius cluster consists of homogenous compute nodes, we can expect every FPGA and CPU to perform a nearly equal number of comparisons per second. Consequently, a straight-forward static load balancing is sufficient to map the application onto the cluster. Portability was achieved by using MPI to implement the parallelized version.

## 4.4 Results

To verify the design presented in Section 4 and determine its performance, we conducted tests with real data generated by the cube cut algorithm. The dataset used consisted of $409'685$ bit strings containing $148 - 160$ ones (list $B$) and $5'339'385$ strings containing $147$ ones (list $A$). Figure 4 shows the resulting runtime. Using one CPU, the software solution finishes after 2736 seconds, while the FPGA takes only 99 seconds. Parallelizing the computations shows the expected nearly linear speedup. A four-node CPU configuration takes 710 seconds, while the same configuration equipped with FPGAs finishes after 26 seconds. To summarize, one hybrid CPU/FPGA node achieves a speedup of 27 over a CPU-only node; a 4-node hybrid cluster is faster by a factor of 105 over a single CPU.

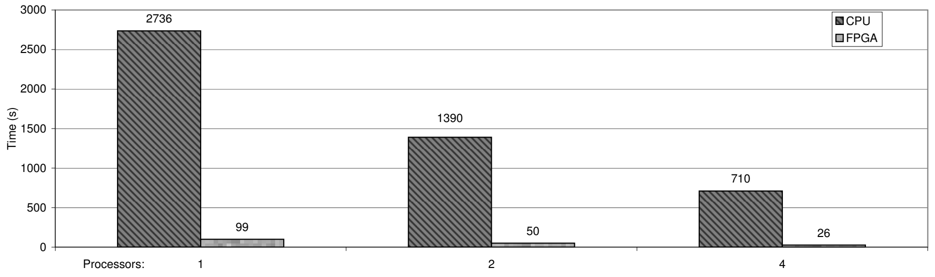


Figure 4. Computation time using CPU vs. FPGA

## 5 Conclusion and Future Work

In this paper we presented an FPGA accelerator that achieves impressive speedups for the most runtime-intense part of the cube cut algorithm. We supported the design process by modeling the architecture and the application. Although the considered application led to a straight-forward accelerator design, the model proved to be very useful for analyzing performance bottlenecks and comparing design alternatives at an early design phase. Further, the model will be invaluable for optimizing the accelerator to different target platforms. We would like to know, for example, what performance we can expect by using a larger FPGA, a faster FPGA, improved DMA controllers, a faster I/O bus, etc.

We synthesized our accelerator design to one of the largest Xilinx FPGAs currently available, the Virtex5 LX330. On that device, we would be able to enlarge the comparator chain to about 400 comparators running at about $116$ MHz. This would increase our speedup by a factor of eight in the best case. Basically, our accelerator is described by the parameters $(n, l, p, m)$, the length of the bit string, the number of parallel pipelines, the numbers of stages per pipeline, and the number of comparisons per pipeline stage. Depending on the speed of the design and the bandwidth of the DMA controllers and the FIFOs, our model allows us to determine the best settings for these parameters.

We believe that creating models of the architecture and the application can greatly facilitate performance estimation and optimization. We have to investigate, however, to what extent the experience with the model for the cube cut problem can be generalized.

Future work on accelerating the cube cut algorithm will focus on the algorithm's third phase. The final composition of the remaining cuts to a minimal set of cuts slicing all edges might be another interesting candidate for FPGA acceleration. Further, we will evaluate the scalability of our design by porting it to different FPGA accelerator boards providing denser FPGAs and an improved transfer bandwidth.

## References

1. C. Sohler and M. Ziegler. Computing Cut Numbers. In *Proceedings of the 12th Annual Canadian Conference on Computational Geometry (CCCG 2000)*, pages 73–79, 2000.
2. András Hajnal, Wolfgang Maass, Pavel Pudlák, György Turán, and Márió Szegedy. Threshold circuits of bounded depth. *J. Comput. Syst. Sci.*, 46(2):129–154, 1993.
3. Egon Balas and Robert Jeroslow. Canonical cuts on the unit hypercube. *SIAM Journal on Applied Mathematics*, 23(1):61–69, 1972.
4. Novikoff. On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622. Polytechnic Institute of Brooklyn, 1962.
5. M.R. Emamy-Khansary. On the cuts and cut number of the 4-cube. In *Journal of Combinatorial Theory Series A 41*, pages 211–227, 1986.
6. P. O'Neil. Hyperplane cuts of an $n$-cube. In *Discrete Mathematics 1*, pages 193–195, 1971.
7. Michael E. Saks. Slicing the hypercube. In Keith Walker, editor, *Surveys in Combinatorics*, pages 211–255. Cambridge University Press, 1993.
8. `http://wwwcs.uni-paderborn.de/cs/ag-madh/WWW/CUBECUTS`.
9. Jens Simon and Jens-Michael Wierum. The latency-of-data-access model for analysing parallel computation. In *Information Processing Letters - Special Issue on Models of Computation*, volume 66/5, pages 255–261, June 1998.
10. Florian Schintke, Jens Simon, and Alexander Reinefeld. A Cache Simulator for Shared Memory Systems. In *Computational Science (ICCS), LNCS 2071*, volume 2074, pages 569–578. Springer, 2001.
11. `http://wwwcs.uni-paderborn.de/pc2/services/systems/arminius.html`.
12. `http://www.alpha-data.com/adm-xp.html`.
13. Rolf Krueger and Brent Przybus. Virtex Variable-Input LUT Architecture. White paper, Xilinx, 2004.