# A Custom Computing System for Finding Similarties in Complex Networks

Christian Brugger*, Valentin Grigorovici*, Matthias Jung*, Christian Weis*,
Christian De Schryver*, Katharina Anna Zweig‡, Norbert Wehn*
*Microelectronic System Design Research Group, University of Kaiserslautern, Germany
‡Graph Theory & Complex Network Analysis Group, University of Kaiserslautern, Germany
{brugger, jungma, weis, schryver, wehn}@eit.uni-kl.de,
valentin.grigorovici@gmail.com, zweig@cs.uni-kl.de

## ABSTRACT

Complex graphs are at the heart of today's big data challenges like recommendation systems, customer behavior modeling, or incident detection systems. One reoccurring task in these fields is the extraction of network motifs, reoccurring and statistically significant subgraphs. In this work we propose a precisely tailored embedded architecture for computing similarities based on one special network motif, the co-occurrence. It is based on efficient and scalable building blocks that exploit well-tuned algorithmic refinements and an optimized graph data representation approach. On chip, our solution features a customized cache design and a lightweight data path that allows the system to perform over 10,000 graph operations per cycle on each chip. We provide detailed area, energy, and timing results for a 28 nm ASIC process and DDR3 memory devices. Compared to an Intel cluster, our proposed solution uses 44x less memory and is 224x more energy efficient.

## 1. INTRODUCTION

Many applications in the big data context are based on fast and reliable identification of so-called *network motifs* in large networks, i.e., those subgraphs whose occurrence is significantly higher than expected in a random graph model. This enables analyzing large-scale biological data in bioinformatics, the analysis of connections in social networks, incident detection, and general graph data cleaning procedures by link assessment [1].

In this work, we consider a special variant of motifs, the so-called *co-occurrence* (*coocc*) which is defined as the number of common neighbors of two nodes in a graph. It can, for example, be used to clean biological high-throughput data or to build e-commerce applications like recommendation systems [1]. However, computing the *coocc* on standard central processing unit (CPU)- and graphics processor unit (GPU)-based architectures is very time and energy consuming.

In this paper we present a dedicated architecture for network motif detection based on the *coocc*. It is problem-independent and universally applicable to a wide application range, for instance as a special accelerator device in bigger system contexts. Due to its modular approach, the proposed design can also be enhanced to other motifs in the future.

Our results show that the proposed architecture clearly outperforms standard CPU server nodes, both with respect to throughput and energy, but also total memory requirements. Compared to one 6-core Intel Xeon X5680 CPU, it is 226x faster with an equivalent power consumption. We demonstrate the performance of our design with the Netflix data set [2, 3] and show that one application specific integrated circuit (ASIC) instance computes the same results in roughly the same time as a 10-node Intel cluster but requires only 2.3% of total memory and less than 0.5% of energy. These superior characteristics allow in particular the use of our architecture in power- and space-limited data-centers and for constructing motif detection systems targeted to process very large graphs with reasonable power consumption and system costs.

In particular, the novel contributions in this paper are:

- We present the first hardware architecture for estimating similarities in graphs with *fixed degree sequence models* [1].
- We introduce algorithms and data paths for *coocc* calculations that massively exploit linear data access patterns.
- We show an efficient cache design that allows the architecture to be scaled without requiring a higher memory bandwidth.
- As results, we give detailed area, energy, and timing results for a 28 nm ASIC process and DDR3 memory devices, including a comparison to an Intel compute cluster.

## 2. BACKGROUND AND RELATED WORK

Everybody who has used social networks knows the famous "Do you also know ...?" function. One could argue that a large part of the success of modern social networks is based on this feature. But how does it work? Which persons should be for instance recommended specifically to you?

A powerful way is to look at the social network itself. There, one can calculate how similar you are to each person in the graph, only using the topology of the connections. Based on that, one can then suggest the top ten most similar people with respect to the direct neighborhood.

In this paper we consider the following method: In the example given in Fig. 1, the similarity between you and Liam is based on the number of common friends you have with her, the so-called *co-occurrence*: $coocc(you, Liam) = 3$. Now we have to ask whether the number three is significant. Assume

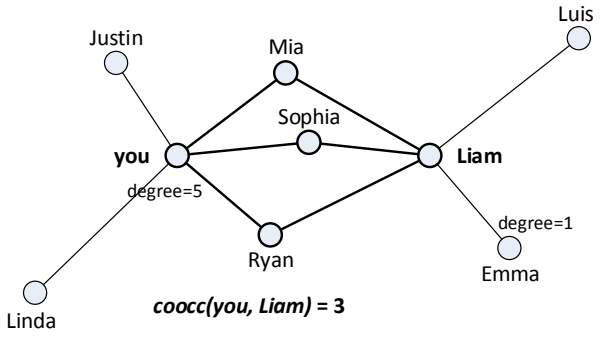


**Figure 1: The *co-occurrence* (*coocc*) between you and Liam is defined as the number of shared friends you have, in this case three.**

you and Liam have thousands of friends, versus you have only three. For this we create random graphs based on the same degree sequence and the premise that nodes have no similarities. In our example the sorted sequence of degrees, the number of edges a node has, is $\{1, 1, 1, 1, 2, 2, 2, 5, 5\}$.

To get the random graphs we swap a sufficient number of pairs of edges, drawn uniformly at random, if and only if no multiple edges would arise due to the swap. This generates independent graphs with the same degree sequence, see Fig. 2. Generating many of such graphs we can calculate the expected *coocc* of this so-called *fixed degree sequence model (FDSM)* [1, 4]. With that information we can judge how significant the *coocc* in our original graph is.

## 2.1 Similarity measures

In general, there exist many ways to quantify the similarity, most of them are various normalization of this *coocc*. In the Jaccard-index, this value is normalized by the cardinality of the joint neighborhood. Pearson's product moment correlation coefficient uses the co-variance of the two vectors, normalized by the product of their standard deviations. However, these classic similarities do not take into account the general distribution of the data. Complex network analysis has shown that in many cases, the number of neighbors is heavily skewed, some authors even suggest that this so-called *degree distribution* is a scale-free distribution [5]. This induces a heavily skewed *coocc* as well in most real-world data sets, which is why the field of complex network analysis suggests to compare the *observed coocc* (number of common neighbors) with the *expected coocc* in an ensemble of random graphs with the same degree distribution [6, 4]. This approach is called FDSM. In this setting, the observed *coocc* can be corrected by the expected *coocc*; the resulting difference is called the *leverage*. The leverage can then be normalized by the standard deviation of the expected distribution to yield the $z$-score [6]. An alternative approach is to use the empirical $p$-value, i.e., the probability to pick a random graph instance in which the *coocc* is at least as high as in the observed network. The usage of an elaborate random graph model to assess the statistical significance of an observed *coocc* by either the leverage, the z-score, or the p-values has shown to produce results with higher quality in many domains and is the basis of this work [7, 1].

## 2.2 Computing leverage, p-value, z-score

Given a bipartite graph $G((V_l, V_r), E)$ with vertices $V_l$ and $V_r$ and edges E, $V_l$ being the side of interest, we define $coocc_i(u, v)$ as the *coocc* for the graph sample $G_i$. Algorithm 1 shows the full scheme. This way of generating graphs is also called *Markov chain Monte Carlo (MCMC)* [8]. The similarity measures for all $u, v \in V_l$ are defined as:

$$coocc(u,v) = \sum_{w \in V_r} \begin{cases} 1, & \text{if } (u,w) \in E \text{ and } (v,w) \in E \\ 0, & \text{otherwise} \end{cases},$$

$$coocc_{FDSM}(u,v) = \text{mean}\left(\{coocc_i(u,v)\}_{i=1,..,|samples|}\right),$$

$$leverage(u,v) = coocc(u,v) - coocc_{FDSM}(u,v), \quad (1)$$

$$p\text{-}value(u,v) = \sum_{i=1}^{|\text{samples}|} \begin{cases} 1, & \text{if } coocc_i(u,v) > coocc(u,v) \\ 0, & \text{otherwise} \end{cases},$$

$$z\text{-}score(u,v) = \frac{leverage(u,v)}{\text{stddev}\left(\{coocc_i(u,v)\}_{i=1,..,|samples|}\right)}.$$

The higher the leverage or the z-score, or the lower the p-value, the more similar the nodes are considered to be. Algorithm 1 shows all the steps, Fig. 2 illustrates the procedure with the friend-recommendation example. For our similarity measures the complexity of Algorithm 1 is $O(|V_l|^2 \cdot |V_r|)$ what makes it very challenging to come up with a scalable implementation.

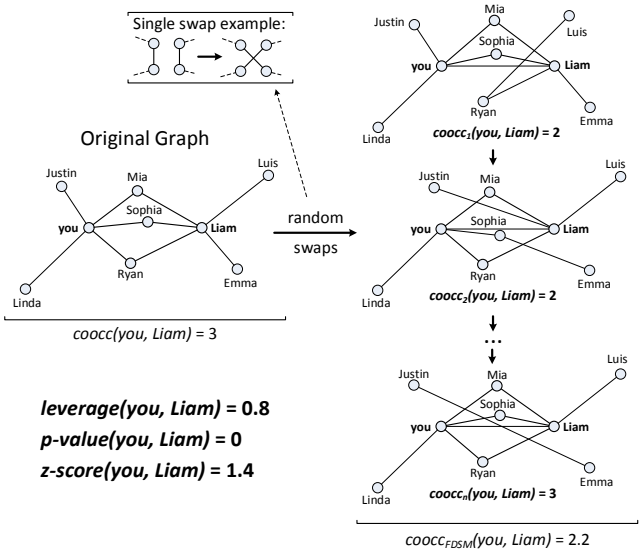


**Figure 2: One hundred random graphs (right) with the same degree sequence as the initial graph (left) obtained by series of 40 swaps. The leverage is the difference of the original *coocc* and the average *coocc* of the random graphs samples (right).**

## 2.3 Related work

Network motif detection is actively investigated in current research, mainly from the algorithmic point of view. From the implementation side, nearly all available work deals with mapping the motif detection problem on parallel CPU and GPU based clusters [9, 10]. To the best of our knowledge there are no publications for accelerating motif detection, in particular by a dedicated accelerator engine optimized for this task. We were only able to detect a few works about hardware implementations for general graph processing, but none of them considering the specific case of *cooccs*.

There are frameworks like GraphGen by Nurvitadhi et al.

**Data**: Graph $G((V_l, V_r); E)$ with vertices $V_l$ and $V_r$ and edges $E$, $V_l$ being the vertices of interest;
**Result**: Leverage, p-value, z-score for all pairs of vertices $(u, v) \in (V_l \times V_l)$;
Calculate $coocc(u, v)\ \forall\ (u, v) \in (V_l \times V_l)$; $G_0 := G$;
**for** $i := 1$ *to* $|samples|$ **do**
  $G_i := G_{i-1}$;
  *Swap randomization:*
  **for** $|swaps|$ **do**
    Choose two edges at random in $G_i$ and swap them, if no duplicate edge arises from the swap;
  **end**
  *Coocc computation:*
  Calculate $coocc_i(u, v)\ \forall\ (u, v) \in (V_l \times V_l)$;
**end**
Calculate leverage, p-value and z-score according (1);

**Algorithm 1:** The complete Link Assessment algorithm, calculating the similarity measures

presented in 2014 [11] or the Graphlet Counting Case Study from Betkaoui et al. in 2011 [12] that generate specific data processing engines for particular graph operations. Both approaches aim at optimizing the memory accessing schemes for the dynamic random-access memory (DRAM) in order to fully exploit the available memory bandwidths. However, they are not application tailored and cover a broad range of graph problems instead of optimizing the performance for motif detection.

# 3. EMBEDDED ARCHITECTURE

Nowadays it is a general trend in high-performance computing (HPC) to enhance current HPC systems with dedicated accelerators that are optimized for time and energy critical tasks what leads to more and more heterogeneous computing architectures. This methodology is adapted from embedded computing where the crucial design points are mainly energy, power, and throughput efficiency. In this paper we present an energy-optimized accelerator architecture for computing similarity measures based on a network-motif-approach in graphs. In detail, it consists of three main parts:

- Swap randomization: generate samples from the FDSM.
- *Coocc* calculation: calculate the *co-occurrence* of all pairs for each random graph sample.
- Similarity measures: generate leverage, p-value, and z-score based on the *coocc*.

Fig. 3 gives an overview of the architecture described next. The global aim of our design is to minimize the number of random DRAM accesses. For that reason, selecting appropriate data structures is the most crucial point. Instead of using adjacency lists, we use an adjacency matrix to store the graph. In that structure, accessing individually nodes directly corresponds to one access, as we exactly know where to look.

Although the graphs we are dealing with are sparse, we are storing them in adjacency matrices and not in adjacency lists. While this might sound surprising, matrices have so many advantages for the architecture that it outweighs the disadvantages by far, as we will see in Section 3.2 and 3.3.

## 3.1 Data structures

We are using three data structures: an adjacency matrix holding the graph, an adjacency list holding the edges, and a result matrix.

Given a bipartite graph $G(V_l, V_r; E)$ consisting of the vertices $V_l$ and $V_r$ and the edges $E$, an adjacency matrix $A = (V_l \times V_r)$ is stored. An entry in the matrix is $A_{u,v} = 1$ if $(u, v) \in E$. It is sufficient to store $A$ with one bit per entry and total storage requirement of $|V_l| \cdot |V_r|$ bit. Furthermore, the edges are stored as a list in $L$. An *edge* is defined as the pair of vertex indices which are connected by it. The edge list requires $|E|(\log_2 |V_l| + \log_2 |V_r|)$ bits.

The result matrix contains partial results for calculating the p-value and z-score for all the pairs in $V_l$. For each pair $(u, v) \in (V_l \times V_l)$ we store an entry in an upper triangular matrix, see Table 1. In a post-processing step, all similarity measures can be efficiently calculated with these partial results on a CPU.

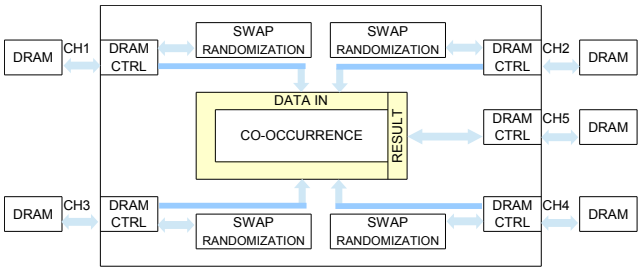


**Figure 3: Overall architecture. The swap randomization blocks and the *co-occurrence* work on datasets residing in each of the DRAMs in a round-robin fashion, while multiple swap randomization blocks can work in parallel. The result is stored separate.**

**Table 1: Partial result matrix entry**

| Variable | Required bits |
|---|---|
| $coocc(u, v)$ | $\log_2(\lvert V_r \rvert)$ |
| $\sum_i coocc_i(u, v)$ | $\log_2(\lvert V_r \rvert \cdot \lvert samples \rvert)$ |
| $\sum_i coocc_i(u, v)^2$ | $\log_2(\lvert V_r \rvert^2 \cdot \lvert samples \rvert)$ |
| p-value count | $\log_2(\lvert samples \rvert)$ |

## 3.2 Swap randomization

**for** $|swaps|$ **do**
  Generate two random numbers (RNs): $a, b \in [1, |E|]$;
  Read the two edges: $u, x := L_a$ and $v, y := L_b$;
  Check existence of swapped edges by reading:
  $k := A_{u,y}$ and $l := A_{v,x}$;
  **if** *both edges do not exist:* $k = l = 0$ **then**
    Swap the edges by writing: $A_{u,x} := 0$; $A_{v,y} := 0$;
    $A_{u,y} := 1$; $A_{v,x} := 1$;
    Update edge list by writing: $L_a := u, y$;
    $L_b := v, x$;
  **end**
**end**

**Algorithm 2:** Implementation of the swap randomization step in Algorithm 1

We sample graphs from the FDSM based on MCMC. A new sample $G_i$ can be generated from the previous one $G_{i-1}$ by randomly swapping $|swaps|$ edges, starting from the original graph. In our architecture, the swapping is realized in a finite-state machine (FSM) implementing Algorithm 2.

For generating the RNs, we use a Mersenne twister (MT)

19937 algorithm [13]. In this formulation, four random reads and six random writes are necessary per swap.

## 3.3 Coocc calculation

Calculating the *coocc* between all pairs of vertices in $V_l$ in a naive way requires to load the same data many times, making memory a bottleneck. For example, calculating the *coocc* between $u, v \in V_l$ requires edges connected to $u$ and $v$, or in other words the two rows $u$ and $v$ of the matrix $A$. When the *coocc* is later calculated between $u$ and $w$, the same row $A_u$ needs to be loaded. This leaves huge potential for an optimized memory hierarchy and algorithms to miminizing data transfer. Solving this is our first major contribution.

First we add a row-cache to the *coocc* module. Second we structure the accesses in such a way that, once a row is loaded into cache, it is not necessary to load it a second time. Furthermore we use multiple modules to parallelize the computations.

Having $k$ parallel *coocc* units, we use their caches to store a consecutive block of $k$ rows $A_u, .., A_{u+k-1}$. Then we stream one by one all following rows through the *coocc* modules, starting with $A_{u+k}$. With each new row $A_v$ the modules can calculate the *coocc* of all pairs of the cached rows $(u, v), .., (u+k-1, v)$. Algorithm 3 formalizes the scheme.

```
Data: Graph G((V_l, V_r); E) stored as adjacency matrix
      A = (V_l × V_r), V_l being of the vertices of interest;
Result: coocc for all pairs of vertices (u, v) ∈ (V_l × V_l);
for u := 1 to |V_l| step K do
    k := 0;
    for v := u to |V_l| do
        Stream row A_v from external memory;
        if k ≥ 1 then
            Compare the streamed row with all
            previously cached rows 1 to k and calculate
            the coocc for the pairs:
            (u, v), .., (u + k − 1, v);
        end
        if k < K then
            k := k + 1;
            Store the streamed row in cache k;
        end
    end
end
```

**Algorithm 3:** Implementation of the *coocc* computation step in Algorithm 1 for $K$ *coocc* modules

Looking closely, you will notice that this scheme also solves the scaling problem. While adding $m$ times more modules reduces the runtime by a factor of $m$, it does not increase the requirements for external bandwidth, still only one row has to be streamed through all the blocks at each given time. This allows us to place hundreds of *coocc* units next to each other, providing massive speedups.

To calculate the actual *cooccs*, we propose efficient data paths, our second major contribution. While standard CPU and GPU architectures only standard data types can be used, on the ASIC we are much more flexible. Assuming that our rows are large, we receive them in blocks of data, $l$ bits per cycle. Having stored $A_u$ in the caches (LMEM) and streaming $A_v$, the $coocc(u, v)$ can be optained by counting all vertices that are connected to both $u$ and $v$. The row $A_u$ at position $w$ is "1", if and only if there is an edge between $u$ and $w$, for $A_v$ equivalently. Counting common edges is equivalent to computing the cardinality of $A_u \& A_v$. This requires a lot of additions.

Fig. 4(a) shows our efficient data path tailored to this task, consisting of an adder tree and accumulator. It is able to process $l$ edges per cycle. In the first stage, it uses only 1 bit adders, 2 bit adders in the second stage, and so on.

## 3.4 Similarity measures

Once all blocks of the row $A_w$ have been streamed, the final *coocc* value is computed. Based on that, all partial results from Table 1 can be updated. This involves reading the entry from memory, updating, and writing it back, see Fig. 4(c). It is worth noticing that this block does not have to operate in each clock cycle, only once per complete row, so that most of the operators can be shared among multiple *coocc* computation blocks. After all samples have been computed, the final similarity measures can be calculated on a CPU by going once over the result data, which only takes a few seconds.

Calculating the initial *coocc* is also performed on chip at the beginning and stored in the partial results.

## 3.5 Parallelization

Parallelization is easily possible by using multiple instances with each instance working on independent samples. For $k$ instances this reduces the total time by a factor of $k$. They start all with the same initial graph and the results can be easily combined at the very end by summation.

# 4. ASIC IMPLEMENTATION

We have implemented our architecture on register-transfer level (RTL) with SystemVerilog and synthesized it with the Synopsys Design Compiler for an advanced 28 nm low-power technology node. Place&Route was performed with the Cadence SoC Encounter. Based on the Netflix data we have extracted realistic activities that we fed into Synopsys PrimeTime to obtain accurate power estimates.

We have integrated three 64 bit DDR3 memory controllers, one for the result and two for storing the graph. At each given time, the random swapping block is operating on one controller and the *coocc* modules on the others. When both are finished they switch over.

In the *coocc* module, the edge caches are 64 kB each, targeting a frequency of 400 MHz. For a 64 bit double data rate (DDR) channel at 800 MHz we get 256 edges per cycle when running the *coocc* units at 400 MHz. That means the adder tree has a width of 128 adders at the top and a depth of seven stages. Pipelining the tree was not required for our application. The operations to calculate the partial results for the similarity are designed with 64 bit for the squares and 32 bit for the rest, being shared over the cell.

We have synthesized four *coocc* modules in a single cell and then combined them in a grid of 5 times 12. In total, our ASIC architecture has 240 *coocc* modules. To distribute the data to the caches or to stream further rows of the matrix a tree-like replication network is used, while for the results a shift register over the whole chip is used. That makes the architecture perfectly scalable. In total, our architecture performs $240 \cdot 256 = 61{,}440$ graph operations per cycle.

The rest of the design is occupied by memory controllers and IO, see Fig. 5. Table 2 shows the resources of each module. For the memory controllers we have estimated the numbers based on the corresponding publications [14, 15]. The whole chip has a size of 51.2 $mm^2$ and an average power consumption of 11.7 W.

We assume that the final system is equipped with two

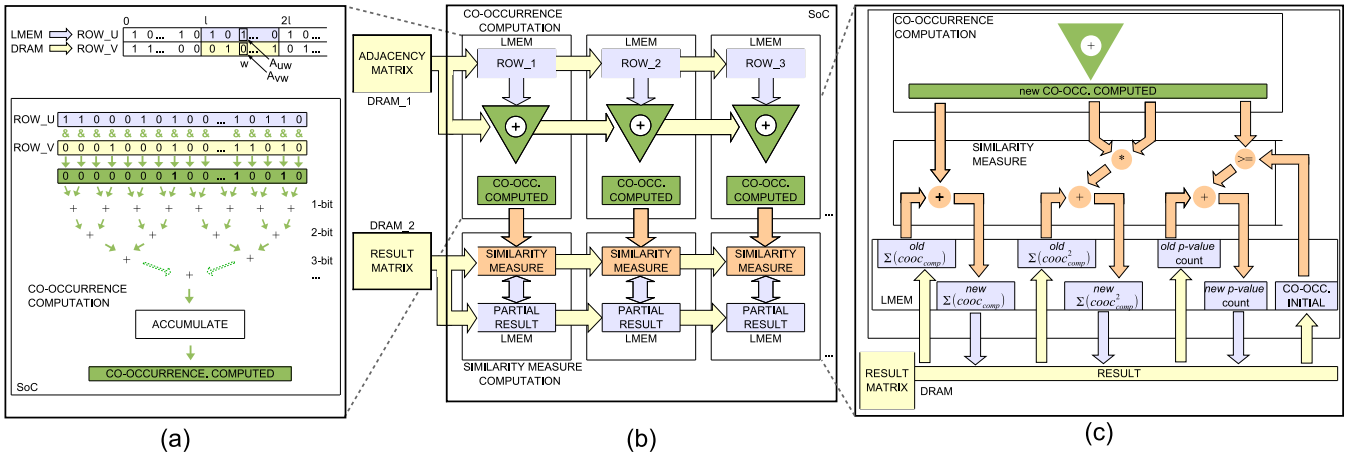


**Figure 4: The *coocc* and result module (b) works on one dataset after another, always updating the same result. It loads one row of the graph into the caches (LMEM) and first calculates the *coocc* before calculating the similarity measures. The *coocc* module (a) consists of an efficient adder tree operating on blocks of $l$ edges per cycle. While the similarity measures (c) consists of several arithmetic blocks it is only called once per row, making it possible to share most of the resources.**

**Table 2: ASIC Ressources**

| Component | Size [mm$^2$] | Frequency [MHz] | Power[1] [W] |
|---|---|---|---|
| Swap randomization | 0.01 | 400 | 0.002 |
| 4 *coocc* cell | 0.572 | 400 | 0.123 |
| DDR3 controller | 4.8 | 800 | 0.8 |
| IO and interconnect | 2.4 | 400 | 0.56 |
| One DDR3 DIMM during: | | | |
| swapping | – | – | 1.8 |
| coocc computation | – | – | 2.2 |
| result processing | – | – | 1.5 |
| Our chip[2] | 51.2 | 400 | 11.7[3] |

[1]fully utilized for single components
[2]240 *coocc* modules, 1 swap module, 3 DDR3 controllers
[3]based on activities running the Netflix dataset

2 GB DDR3-1600 DIMMs for the graphs and a 4 GB module for the result. We have modeled the timing of the DIMMs with DRAMSys tool [16] and estimated power with DRAMPower tool [17]. As inputs to the tools we created DRAM access patterns for all the modules, specifying in which order, when, and at which address the DRAM is accessed. Based on those trace files the tools provided us with exact timings and power estimations given in Table 2.

Note that neither the architecture nor the algorithm makes any assumption about the properties of the graph: Every graph can be processed by the ASIC. Of course as in every computing architecture the graph needs to fit into system memory. The number of *coocc* units was chosen such that the chip area is around 50 mm$^2$, a reasonable target for today's IP-cores. At runtime, the input graph is folded in space and time onto the available *coocc* units and buffers.

## 5. CLUSTER COMPARISON

To demonstrate the performance of our design we have calculated the similarity measures for the Netflix dataset [2, 3]. Netflix, a commercial video streaming service, has released 100,480,507 user ratings for all of their 17,700 movies from 480,189 users. While users give ratings in the range of 1-5, we have extracted an input graph with edges between users and movies whenever the rating is 4 or 5 only. As a result, the input graph has 17,769 movies, 478,615 users, and 56,919,190 edges. In this case $V_l$ will be the movies, $V_r$ the users, and $E$ the 4, 5 ratings.

The swap randomization module of our design takes 2.14 s to generate a new random graph by swapping the dataset with $|swaps| = |user| \log |user| = 6{,}259{,}639$. The *coocc* module, working in parallel, takes 3.25 s to calculate the *coocc* partial results from one graph. During this time the result memory controller is active in 20 % of the time. We have used a total number of samples $|samples| = 10{,}000$ that ensure sufficient convergence. In total, our design takes 9.0 hours to process the Netflix data.

For a system level comparison, we have included the power of our design and DRAM as well as a 20 % overhead of 2.63 W to account for necessary board components (ethernet, clocks) and the power supply. Post-processing and calculating the final similarity measures for the Netflix data takes on an Intel node 0.325 seconds or 120 J, during the rest of the time the CPU is free to use for other purposes.

To make a fair comparison, we have spent a lot of effort on optimizing the parallel cluster implementation. Two skilled persons spent three months full time for this reference work. Optimization involved selection of algorithm that minimize computing time for the given memory resources, removing locks by data partitioning and data access linearization. Since swapping is hard to parallelize, each core works on its own sample during swapping, generating 12 samples on one server node. Afterwards, the partial results are updated one sample after another, while all the 12 cores work in parallel on one sample to reduce memory requirements. Among nodes, the parallelization is the same as for the chips with each node working on independent samples. Communication is performed over InfiniBand only required to distribute the data at the beginning and aggregate the results at the very end of the algorithm.

For the cluster implementation, the swapping works on the same adjacency matrix $A$ as in the proposed architecture to minimize random accesses. At the same time an

**Table 3: Cluster ASIC comparison**

| Implementation | Memory[1] [GB] | Runtime [hour] | Power [W] | Energy [MJ] |
|---|---|---|---|---|
| 10 node Intel cluster[2] | 202/480 | 8.5 | 3700 | 114 |
| This work[3] | 4.6/8 | 9.0 | 15.8 | 0.51 |
| Improvement | 44x | 0.95x | | 224x |

[1]used/available memory
[2]each node: 2×Intel Xeon X5680 @ 12×3.33 GHz, 32 nm; 48 GB DDR3 memory
[3]node inclduing: ASIC with 240 modules, 28 nm; 8 GB DDR3 memory; board (ethernet, clocks), power supply.

adjacency list is kept in memory that contains the user ids of the people who have rated the film for each movie. This adjacency list is used to calculate the *coocc*, since using the matrix for *coocc* calculations is very inefficient on the CPU.

For the cluster we have measured the average system power with a power meter during operation. Note that the power measurements for both the cluster and proposed architecture consider the entire link assessment algorithm, assuming the input and result are in memory. The results are listed in Table 3.

# 6. CONCLUSION

Discovering similarities in graphs is an important task in many big data applications. However, for large graphs this is in general a very time and memory consuming job on standard computing clusters. In this work we introduce a dedicated hardware architecture that implements the three basic steps for this tasks: swap randomization, *coocc* computation, and calculating the similarity measure. To the best of our knowledge, it is the first tailored hardware architecture for computing similarity measures based on a network-motif-approach in graphs. Our design is universally applicable to a large range of big data applications like bioinformatics, recommendation systems, friend suggestions in social networks, or general graph cleaning as it is. Due to its modular structure, it can be easily enhanced to other network motifs and similarity measures.

Since the main limitation in this application is in general the memory bandwidth utilization to the DRAM, we have optimized our architecture for minimal DRAM accesses. For that purpose, our proposed design features well-matched data structures: Instead of using adjacency lists as normally done in CPU clusters, we implement the complete adjacency matrix, but with 1 bit entries only. This is not practical in generic computing platforms in this way. With that approach we minimize the data transmission in general and in particular the DRAM bandwidth.

We have synthesized our design for a 28 nm process and could achieve a clock rate of 400 MHz for a die size of 51.2 $mm^2$ and a power consumption of 11.7 W.

As a result, the proposed architecture is both energy and memory efficient. In comparison to a 10-node standard dual-socket Intel cluster, our design achieves the same throughput with less than 0.5% of power and therefore energy per task. At the same time, it requires only 2.3% of DRAM.

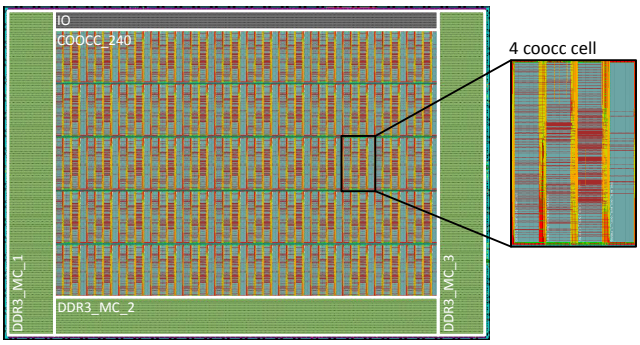


**Figure 5: ASIC floorplan of our 28 nm chip. The chip consists of 240 *coocc* modules, three DRAM controllers and IO logic. The swap randomization block is not visible here due to its small size.**

# 7. REFERENCES


[1] Katharina Anna Zweig et al. *A systematic approach to the one-mode projection of bipartite graphs*. Social Network Analysis and Mining, 1(3):187–218, 2011.

[2] http://www.netflixprize.com/, last access: 2014-12-01.

[3] James Bennett et al. *The netflix prize*. In Proceedings of KDD cup and workshop, volume 2007, page 35, 2007.

[4] Aristides Gionis, et al. *Assessing data mining results via swap randomization*. ACM Transactions on Knowledge Discovery from Data, 1(3):article no. 14, 2007.

[5] Albert-László Barabási et al. *Emergence of Scaling in Random Networks*. Science, 286(5439):509–512, 1999.

[6] Ron Milo, et al. *Network Motifs: Simple Building Blocks of Complex Networks*. Science, 298:824–827, 2002.

[7] Emőke-Ágnes Horvát, et al. *A network-based method to assess the statistical significance of mild co-regulation effects*. PLOS ONE, 8(9):e73413, 2013.

[8] George W. Cobb et al. *An Application of Markov Chain Monte Carlo to Community Ecology*. The American Mathematical Monthly, 110:265–288, 2003.

[9] Pawan Harish et al. *Accelerating Large Graph Algorithms on the GPU Using CUDA*. In Srinivas Aluru, et al., editors, High Performance Computing (HiPC), volume 4873 of Lecture Notes in Computer Science, pages 197–208. Springer Berlin Heidelberg, 2007.

[10] B.A. Miller, et al. *A scalable signal processing architecture for massive graph analysis*. In Proceedings of the 2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5329–5332, March 2012.

[11] Eriko Nurvitadhi, et al. *GraphGen: An FPGA Framework for Vertex-Centric Graph Computation*. In Proceedings of the 2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 25–28, May 2014.

[12] B. Betkaoui, et al. *A framework for FPGA acceleration of large graph problems: Graphlet counting case study*. In Proceedings of the 2011 International Conference on Field-Programmable Technology (FPT), pages 1–8, Dec 2011.

[13] Makoto Matsumoto et al. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. ACM Trans. Model. Comput. Simul., 8(1):3–30, January 1998.

[14] Jason Howard, et al. *A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS*. In Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, pages 108–109. IEEE, 2010.

[15] Denis Dutoit, et al. *A 0.9 pJ/bit, 12.8 GByte/s WideIO memory interface in a 3D-IC NoC-based MPSoC*. In VLSI Technology (VLSIT), 2013 Symposium on, pages C22–C23. IEEE, 2013.

[16] Matthias Jung, et al. *TLM modelling of 3D stacked wide I/O DRAM subsystems: a virtual platform for memory controller design space exploration*. In Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[17] Karthik Chandrasekar, et al. *Improved Power Modeling of DDR SDRAMs*. In proc. DSD'11, 2011.