A Custom Hardware Architecture for the Link Assessment Problem

André Chinazzo(✉), Christian De Schryver, Katharina Zweig, and Norbert Wehn

TU Kaiserslautern, Kaiserslautern, Germany
{chinazzo,schryver,wehn}@eit.uni-kl.de, zweig@cs.uni-kl.de

Abstract. Heterogeneous accelerator enhanced computing architectures are a common solution in embedded computing, mainly due to the constraints in energy and power efficiency. Such accelerator enhanced systems dispatch data- and computing-intensive tasks to specialized, optimized and thus efficient hardware units, leaving most control flow tasks for the more generic but less efficient central processing units (CPUs). Nowadays, also high-performance computing (HPC) systems are becoming more heterogeneous by incorporating accelerators into the computing nodes.

In this chapter, we introduce the concept of heterogeneous computing and present the design of a hardware accelerator for solving the Link Assessment (LA) problem, in introduced Chapter 3. The hardware accelerator integrates its main dedicated processing units with a customized cache design and light-weight data path. We provide detailed area, energy, and timing results for a 28 nm application specific integrated circuit (ASIC) process and DDR3 memory devices. Compared to an CPU-based cluster, our proposed solution uses 38x less memory and is 1030x more energy efficient for processing a users-movies dataset with half a million edges.

Keywords: Link assessment · Application specific · Custom hardware · DRAM

1 Introduction

Nowadays, we live in the era of the so-called *data deluge*, i.e., the increase in produced data supersedes the progress in the available compute performance. This poses heavy challenges on data-centric (statistical) methods, algorithms, and compute systems [18]. Among others, selecting the appropriate data structures, heterogeneity, and parallelization schemes are crucial for achieving high computing performances with low energy demands. For example central processing unit (CPU)-based systems can only access data stored in memory as complete words (cache lines) and work with fixed data types. In contrast, dedicated hardware accelerators allow custom bit widths and data types. This can not only save energy due to avoiding unnecessary data transfers and operations but also allowing direct bit-wise operations like, e.g., accessing one-bit-column entries in a matrix.

H. Bast et al. (Eds.): Algorithms for Big Data, LNCS 13201, pp. 57–75, 2022.
https://doi.org/10.1007/978-3-031-21534-6_4

In general, standard computing architectures based on CPUs and graphics processor units (GPUs) are moving data around heavily. However, in modern technologies, data transfers and storage in general consume much more power than the actual computing [5]. In particular, accessing (off-chip) dynamic random-access memory (DRAM) is a very time- and energy-consuming task. This leads to the concept of the so-called *data-driven* or *dataflow computing*, e.g., employed in the Google TensorFlow architecture [5]. Such architectures focus on the data stream and manipulate data on-the-fly, avoiding unnecessary storage and data transfers.

In addition, in data centers, servers alone only consume around one-third of the total power, while the rest is required for cooling, communication, storage, and building supply [8]. Seen from a different perspective, the maximum available power budget of a system (or a data center) is a hard limit for the available computing power. The latter can only be increased by installing compute systems with a higher power efficiency (e.g., incorporating special hardware accelerators, for instance with a dataflow architecture). Thus, reducing the power demand of the compute servers in combination with the smart reduction of inter-server communication can lead to a total of 2-3x power savings in the data center itself.

Modern system on chips (SoCs) in the mobile, embedded, and Internet-of-Things (IoT) domain are heavily heterogeneous systems with plenty of custom components for dedicated purposes such as audio decoding, video en- and decoding, radio transmission, or sensor data pre-processing in a mobile phone. In particular for mobile devices, there are hard limits for both energy (battery capacity) and power (maximum heat dissipation). However, over the last decades we see more and more heterogeneity also in the data centers [1,5]. Examples are general purpose graphics processor units (GPGPUs), the Intel Xeon Phi accelerator cards, or the field programmable gate array (FPGA)-based Amazon EC2 F1 instances released in 2017[1]. One of the major reason is the so-called *Dark Silicon* phenomenon: In modern chip technologies, only a small amount of transistors can be active at a time in order to avoid overheating (and thus destruction) of the device [7]. This also poses a heavy challenge for the classical multi-core approach - more cores of the same type do not provide more computation power if they cannot be powered up all at the same time.

Nevertheless, end-users are not at all interested in the underlying technology of the *services* they use. Nowadays, most services are distributed over an information technology (IT)-infrastructure from IoT nodes, mobiles, edge servers, and data centers [13]. Thus, the overall application is partitioned and disseminated on various parts of the IT-infrastructure, all with probably different computing architectures and characteristics. As an example, consider a real-time navigation service from Google or Apple: The Global Positioning System (GPS) coordinates collected by (maybe external) GPS receivers are sent to the SoC of the mobile that acts as a human-machine interface (HMI), displaying the route. However, the route itself is calculated in a data center of the service provider. In addition, GPS data from other service users is employed for estimating traveling times and traffic jams, and incorporated in the route calculation.

[1] See https://aws.amazon.com/ec2/instance-types/f1/. Last accessed on 24/11/2022.

In this chapter, we give an overview of hardware-assisted compute systems for applications based on the *Link Assessment (LA)* algorithm. The LA algorithm can be used to clean up large network data sets with noisy data. It assesses the structural similarities between the nodes, and thus differentiates meaningful relationships between nodes from noisy ones [19 SPP]. The LA algorithm as presented in Chapter 3 can be employed on a large scale of applications, e.g., recommendation systems, protein-protein interaction analyses in biology, or business analytics and marketing [3 SPP].

In Sect. 2 we give a short overview about the fundamentals of hardware (HW) and hardware/software (HW/SW) design both for custom application specific integrated circuit (ASIC) and FPGA architectures. Section 3 provides detailed insights in our proposed HW architecture for the Link Assessment (LA) algorithm. Performance data and comparisons are given in Sect. 4. Section 5 concludes this chapter.

2 Basics of Hardware and Systems Design

Custom, dedicated hardware compute architectures are substantially different from standard programmable architectures such as CPUs or GPUs. They are tailored for a specific task, avoiding all unnecessary overhead in storing/moving data, for control architectures, and over-precision data types. This increases both compute performance and power/energy efficiency, at the cost of low to zero flexibility after design. In contrast to a program written for CPUs, hardware architectures, in general, do not receive and execute instructions. Instead, their behavior is encoded in the circuit itself.

Hardware accelerators are electrical (abstracted: digital) circuits that focus on data manipulation. They can be realized in three ways:

- As circuits with various discrete components on a printed circuit board (PCB),
- As a fixed geometry on silicon (a so-called *application specific integrated circuit (ASIC)*), or
- On an underlying configurable hardware architecture such as a *programmable logic device (PLD)*, in particular an *field programmable gate array (FPGA)*.

Nowadays, most systems are realized on a so-called *system on chip (SoC)*. In contrast to discrete circuits realized on PCBs, a SoC combines most components on a single piece of silicon. For that purpose, various *processing elements (PEs)* are attached to a communication infrastructure (a bus or a *network on chip (NoC)*). In addition, external input/output (I/O) interfaces are provided for receiving from and sending data to the outside world. An example for such a SoC structure is given in Fig. 1.

In general, not all PEs are developed by the system designer (team) on their own. Instead, many component architectures are available for purchasing as so-called *intellectual property (IP)*, i.e., as hardware geometry or as design data given in a hardware description language (HDL) or a logical netlist. They mostly

ship with an equivalent software model that can be used for behavioral analysis, testing, and debugging purposes. IP cores can somehow be compared to software libraries in programming since they offer predefined functionalities that can be incorporated into the overall systems. However, most IP cores are closed-source and only available on a commercial basis. In contrast to software projects, open-source hardware platforms such as *opencores.org* are very limited, both from their available contents and their technology.

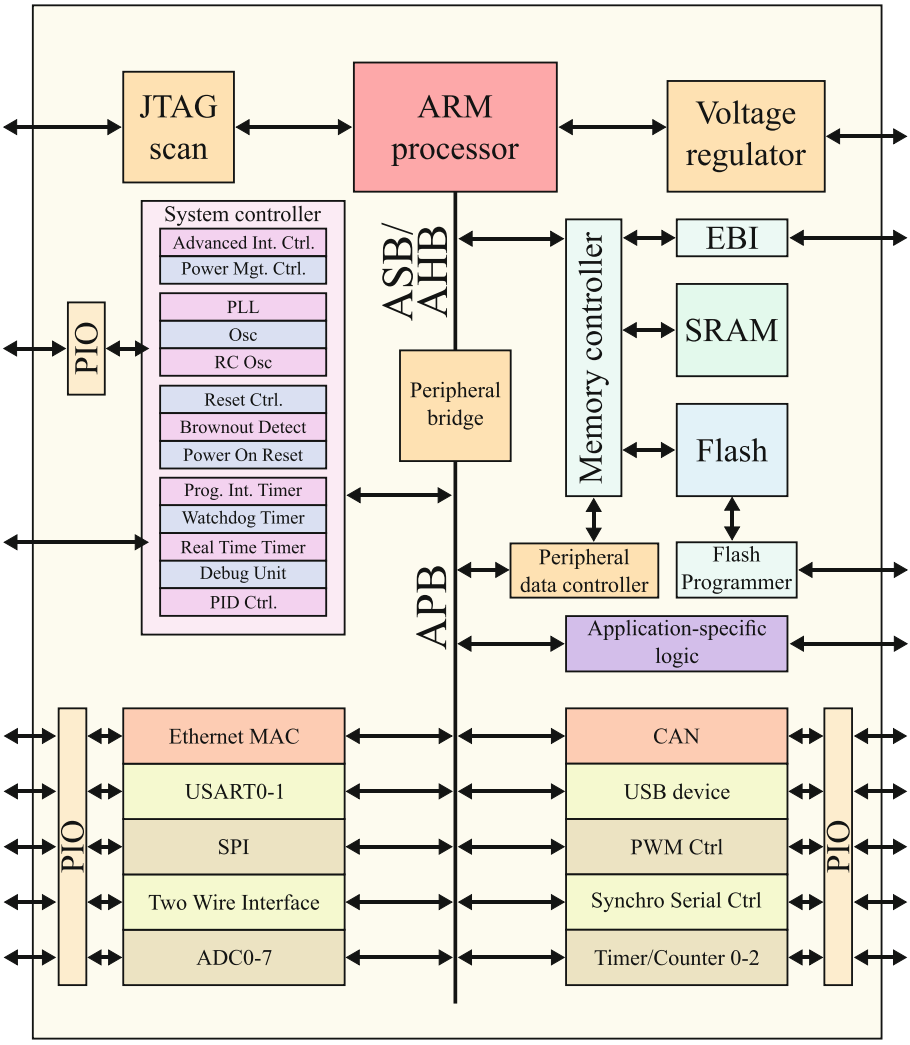

Fig. 1. Example for a SoC with processing elements, interconnect, and interfaces (By en:User:Cburnett - Own work in Inkscape based on en:Image:ARMSoCBlock Diagram.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2866881)

2.1 Hardware/Software System Design Flow

The generic (classic) *design flow*[2] for custom computing systems is shown in Fig. 2. It is much more complex than a pure software development flow. The flow starts with a so-called *hardware-software-partitioning* that determines which parts of the overall behavior will be realized in hardware or software. While considering available hardware and software IP in conjunction with functional and non-functional requirements such as throughput, energy/power limitations, or quality aspects, the system (architecture) platform is determined. After a preliminary simulation, the actual implementation of the hardware and software components starts. Finally, the system components, their interaction, and the final system behavior are validated.

Since we expect software development flows to be well-known by the readers of this chapter, we will focus on the hardware development part in the following.

2.2 FPGA Basics

Hardware architectures realized in an *application specific integrated circuit (ASIC)* can no longer be changed after production (they are fixed geometries in silicon). In contrast, a *programmable logic device (PLD)* is shipped as a device with plenty of available hardware units that can be connected after production. This *programming* or *configuration* can be either one-time[3] or multiple times. A prominent example for the latter is a *field programmable gate array (FPGA)*.

FPGAs are hardware devices that come with a large amount of flexible small hardware units, so-called lookup tables (LUTs). They are basically very small random access memorys (RAMs) that are written during the boot process (*"configuration"*) of the FPGA. Besides, FPGA provide a complex and flexible interconnect system that is configured together with the LUTs. Furthermore, special components such as Block RAMs (BRAMs), fixed bitwidth multiply-accumulate (MAC) units, multipliers, and I/O components are available.

FPGAs do not have a functional behavior before being initially configured. Some types can even be (partially) re-configured during operation, i.e., changing (parts of) the circuit while the rest of the system continues running. Thus, systems equipped with FPGAs allow a very high level of flexibility and dynamics (however, at the cost of an immensely complex design flow, see Fig. 2). In addition, combined CPU/GPU-FPGA systems are available, both in the high-performance computing (HPC)/data center and the embedded SoC domain.

The acquisition of the FPGA vendors Altera by Intel in 2015 and Xilinx by AMD in 2020 shows the potential of this technology for the future of the computing landscape.

[2] A lot of different elaborate system design flows exist [2,11,17] that are omitted here for the sake of clarity.

[3] One-time programmable devices are physically modified during the programming, e.g., by burning connections or melting so-called *antifuses* that create a conducting connection afterwards.

The proposed hardware architecture for computing the Link Assessment (LA) algorithm can be realized both on ASICs and FPGAs. In the following, we present our architecture in detail and illustrate the differences compared to classical CPU implementations.

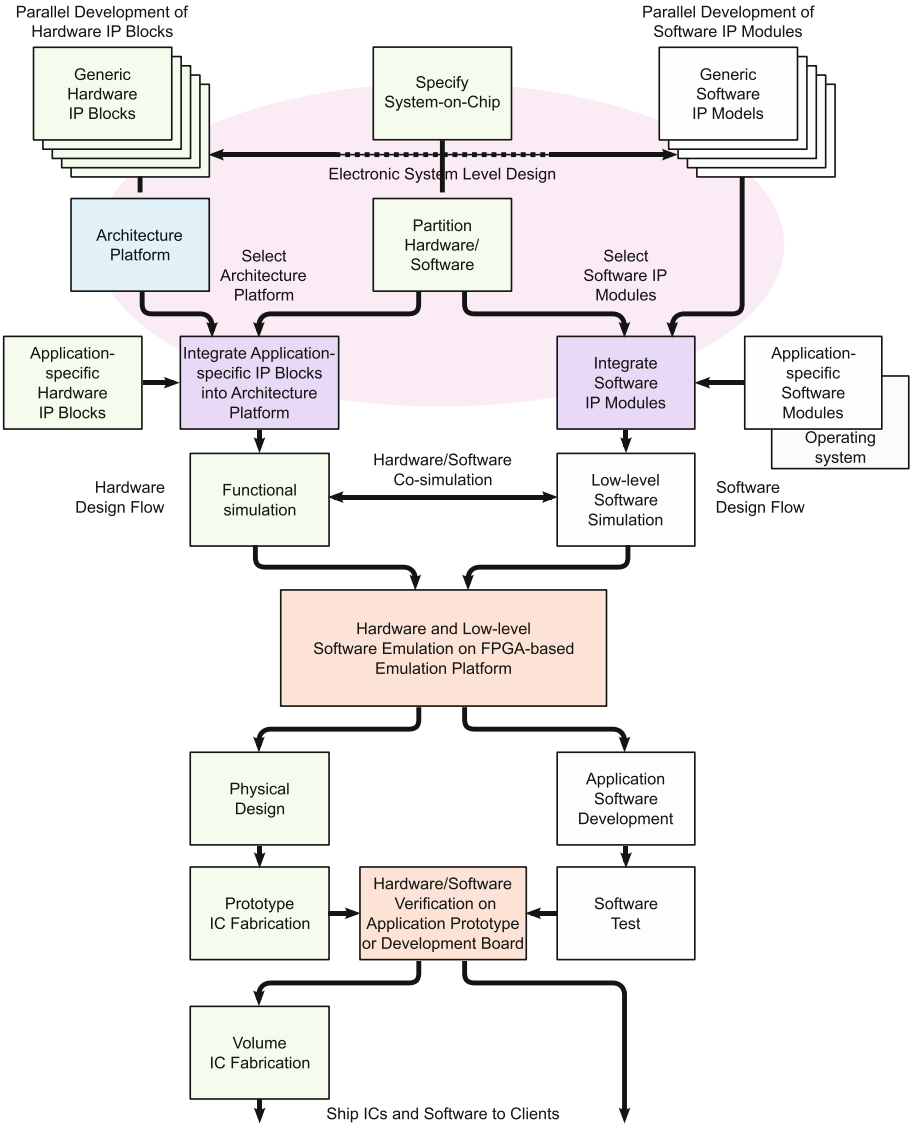

Fig. 2. Generic design flow for a SoC (By Traced by User:Stannered - en:Image:SoCDesignFlow.gif, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1864027)

3 Hardware Architectures for the Link Assessment Computation

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 [15]. This enables analyzing large-scale biological data in bioinformatics, connections in social networks, incident detection, and general graph data cleaning procedures by LA [22 SPP].

Network motif detection is actively investigated in current research, but 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 [9,14].

For the Link Assessment (LA) algorithm, we consider a special variant of motifs, the so-called *co-occurrence (coocc)* which is defined as the number of common neighbors between two nodes of graph. Formally, $coocc\ (u, v) = |N(u) \cap N(v)|$ for any pair of nodes $u, v \in G$, where $N(u)$ is the neighborhood of node u in graph G. Throughout this chapter, we use the shorthand "*coocc* matrix of a network/graph" in place of "the set of all node-pairwise *cooccs* of a network/graph," or $coocc(G) = \{coocc(u, v)\ \forall u, v \in G\}$. For a bipartite graph, $G = G(V_l, V_r; E)$, with vertex partitions V_l and V_r and edges $E \subset (V_l \times V_r)$, the *coocc* matrix can be defined for either partition, e.g., $coocc(V_l) = \{coocc(u, v) \forall (u, v) \in (V_l \times V_l)\}$, in which case V_l is called the side of interest. In this chapter, we focus on bipartite graphs.

The $coocc(u, v)$ by itself is a way of quantifying the similarity of nodes u and v. However, it is a strongly biased quantifier, e.g., w.r.t. the degree of the nodes. The LA algorithm reduces such biases by comparing the observed *coocc* of the real network with its expected value for a random graph model (null-model), namely the fixed degree sequence model (FDSM) [22 SPP,19 SPP]. As the name suggests, the FDSM is the set of all graphs configurations that share the same degree sequence as the observed graph, and it has been shown to provide more robust results than simpler null-models [22 SPP]. Since closed-form solutions for the expected co-occurrences, $coocc_{FDSM}(u, v)$, are not known, these quantities are estimated by a random sampling procedure, known as a Markov chain Monte Carlo (MCMC) approach.

The MCMC approach is divided in two main steps: (1) the randomization of the graph by repeatedly swapping its edges until an uncorrelated, and hence unbiased sample of the FDSM is reached, and (2) the computation of the sample's *cooccs*. Of key importance are (a) the number of swap trials between samples and (b) the number of samples drawn from the FDSM. For the interested reader, Chapter 3 presents the LA in more detail, including an in depth analysis of the effect of those parameters, (a) and (b), on the final quality of the results as well as on the total runtime of the algorithm. In fact, MCMC sampling is the most time consuming part of the LA algorithm.

Once enough samples have been created and evaluated, the node-pairwise similarities are calculated as the probability of finding, in the FDSM, a

coocc(u, v) greater or equal than that of the original graph. The higher the probability, the lower the similarity between (u, v). The probability is estimated first by the p-value and ties are broken by the z-score (see Chapter 3).

In Sect. 3.2 we show that the LA performance is strongly bounded by the speed of the random accesses to the main memory. Aiming to reduce the effects of this unavoidable constraint, in 2015 we have presented the first dedicated embedded hardware accelerator optimized for this task [4 SPP]. Precisely tailored cache memories and computational units for the *coocc* calculation help reducing the number of random accesses by using a rather naive representation of the graph, which is not optimal for CPUs. This work is the basis for a granted patent [21 SPP].

In a follow-up work [3 SPP], we exploit the granularity of DRAM devices to increase the efficiency of main memory accesses during the random graph creation (the null model). We demonstrate the performance of our design with the Netflix Prize data set[4] and show that a single ASIC instance has a speedup of 5.6x compared to a 10-node Intel cluster while requiring 38x less memory and 1030x less energy.

3.1 Data Structures

The Link Assessment (LA) requires two main pieces of information: The graph and the co-occurrence and similarity measures matrices.

The graph is used by both compute kernels, i.e., the edge swapping (see Chapter 2) and the *coocc* calculation. The edge swapping kernel consists of randomly selecting two edges, (u, w) and (v, x) for $u, v \in V_l$ and $w, x \in V_r$, and swapping their connections, to get (u, x) and (v, w), if this does not modify the degree sequences of V_l and V_r. For the edge swapping to have a constant compute complexity, the data structures must provide direct access to existing edges of the graph (random edge selection) and a constant time check for the existence of the new, swapped edges (to preserve the degree sequences). While the adjacency list representation of the graph solves the first task, its adjacency matrix solves the second. Using only one of the data structures would drastically slow the edge swapping procedure. Therefore, we make use of both graph representations, as formalized next.

Given a bipartite graph $G(V_l, V_r; E)$ consisting of the vertex partitions V_l and V_r and the edges $E \subset (V_l \times V_r)$, an adjacency matrix $A = (V_l \times V_r)$ is stored. An entry in the matrix is $A_{u,w} = 1$ if $(u, w) \in E$, with nodes $u \in V_l, w \in V_r$. It is sufficient to store A with one bit per entry and a total storage requirement of $|V_l| \cdot |V_r|$ bits. The adjacency list representation is simply the list of all edges E, requiring $|E|(\lceil \log_2 |V_l| \rceil + \lceil \log_2 |V_r| \rceil)$ bits.

One *coocc* half-matrix is necessary for storing the real graph *cooccs*. It is a half-matrix since *coocc*(u, v) = *coocc*(v, u), and each pair of nodes $(u, v) \in (V_l \times V_l)$ must be evaluated. A second and identical structure is necessary for

[4] Available at https://www.kaggle.com/netflix-inc/netflix-prize-data. Last accessed on 24/11/2022.

storing the *cooccs* of each random graph sample. Instead of keeping as many *coocc* half-matrices as the number of samples, the similarity measures, p-value and z-score, are updated after each sample. For the p-values, a single half-matrix is required. For updating the z-score, it is sufficient to keep the sum and the sum-of-the-squares of the samples' *coocc*.

A summary of the memory footprint of each data structure is shown in Table 1.

Table 1. Memory footprint of the data structures for the LA

Variable	Required bits
Adj. matrix	$\lvert V_l\rvert \cdot \lvert V_r\rvert$
Adj. list	$\lvert E\rvert(\lceil \log_2 \lvert V_l\rvert \rceil + \lceil \log_2 \lvert V_r\rvert \rceil)$
$coocc_{ori}(u,v)$	$\lceil \log_2(\lvert V_r\rvert) \rceil$
$coocc_i(u,v)$	$\lceil \log_2(\lvert V_r\rvert) \rceil$
p-value count	$\lceil \log_2(\lvert samples\rvert) \rceil$
$\sum_i coocc_i(u,v)$	$\lceil \log_2(\lvert V_r\rvert \cdot \lvert samples\rvert) \rceil$
$\sum_i coocc_i(u,v)^2$	$\lceil \log_2(\lvert V_r\rvert^2 \cdot \lvert samples\rvert) \rceil$

3.2 Memory Boundedness

In order to demonstrate the memory boundedness of the LA, we use the roofline model [20] to profile a parallel, optimized CPU implementation of the algorithm. The roofline model is a visualization tool intended to evaluate the efficiency of computation kernels w.r.t. the underlying hardware. The maximum performance of the hardware is bounded, of course, by its maximum number-crunching speed, but also by the memory access bandwidth. These bounds are represented by the black lines (the Rooflines) in Fig. 3. The performance of a computing kernel is measured in operations per second, i.e., how busy the processor really is. Only integer operations (INTOP) are considered because the LA does not use floating-point numbers, and the G in GINTOP stands for Giga, i.e.,, billions of integer operations. The arithmetic intensity is defined as ratio between the number of operations over the total memory traffic, being measured in operations per byte.

The performance and arithmetic intensity of the edge swapping and the *coocc* computation kernels were measured by Intel Advisor[5]. They are presented in Fig. 3. We can see that the performance of the kernels are 1.3 GINTOP/s for edge swapping and 2.2 GINTOP/s for the *coocc* calculation. This is far from the attainable value by the CPU (109 GINTOP/s). This is because of the low arithmetic intensity of both kernels, as is expected from their tasks. The edge swapping kernel, for example, needs to access multiple random memory locations to only check for the existence of an edge, hence many bytes are accessed but

[5] https://software.intel.com/content/www/us/en/develop/articles/intel-advisor-roofline.html.

very little processing happens. Most of the time, this kernel is simply waiting for the data to be loaded, what we call a memory stall. During the stall, no processing occurs.

The impact of the stalls on the total runtime are given per memory hierarchy level. We can see that more that half of the total runtime is spend waiting for the DRAM. Moreover, the DRAM stalls account for almost 80% of the edge swapping runtime. This is expected from the intrinsically random memory access pattern of the edge swapping, which means that the cached data is hardly ever used.

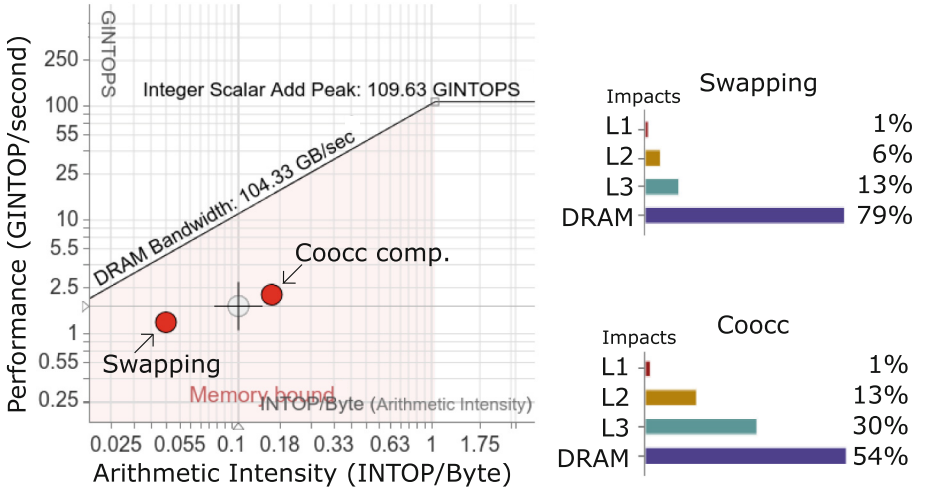

Fig. 3. Roofline analysis of the main compute kernels for the Link Assessment: Edge swapping and co-occurrence computation. Both kernels are strongly memory bounded, with 79% and 54% of the runtime spent in DRAM stalls for the edge swapping and *coocc* kernels, respectively. Machine: Intel Xeon E5-2640 v3 (16 cores at 2.6 GHz) with 2×32 GB DRAM.

3.3 Co-occurrence Calculation

Calculating the node-pairwise *coocc* of a given graph is the most time consuming part of the LA. Using the adjacency matrix, we iterate through each pair of rows (nodes in V_l) and count the number of columns (nodes in V_r) where both elements are 1, i.e., both edges exist. The computational complexity of this procedure is, therefore, $O(|V_l|^2 \cdot |V_r|)$.

Through the adjacency list, the complexity can be amortized to $O(\sum_{V_r} deg(w)^2)$, where $deg(w)$ is the degree of node $w \in V_r$. This particularly benefits networks whose degrees follow a power-law distribution, as is the case of most real networks [22 SPP]. For a CPU implementation of the LA, the adjacency list approach is preferred, even though the memory access pattern is unstructured (see Sect. 3.2).

From a hardware architecture design perspective, however, the adjacency matrix approach can be easily implemented with blocks of bit-wise ANDs followed by an adder tree, what we call *coocc* module. Due to the small size of such an operational block, it can be replicated multiple times, reaching a degree of parallelism that is not feasible in CPUs. To make use of such high parallelism without being constrained by the DRAM bandwidth requires a well-designed cache layout.

Calculating the *coocc* between all pairs of vertices in V_l in a naive way requires to load the same data many times. 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 minimizing data transfer.

We presented an appropriate solution for this issue in 2015 [4 SPP]: The key idea was to add a row-cache to the *coocc* module. The row-cache must be able to store one complete row of the adjacency matrix.

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 1 formalizes this scheme.

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

Data: Graph $G((V_l, V_r); E)$ stored as adjacency matrix $A = (V_l \times V_r)$, V_l being the side of interest
Result: coocc for all pairs of vertices $(u, v) \in (V_l \times V_l)$

```
1 for u := 1 to |V_l| step K do
2     k := 0
3     for v := u to |V_l| do
4         Stream row A_v from DRAM
5         if k ≥ 1 then
6             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)
7         if k < K then
8             k := k + 1
9             Store the streamed row in cache k
```

The main advantage of this scheme is solving 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 since only one row has to be streamed through all the blocks at each given time. This allows us to place hundreds, if not thousands, of *coocc* units next to each other, providing massive speedups.

Figure 4(a) shows the data path tailored to this task, consisting of an adder tree and accumulator. Each edge cache has a capacity of 64 kB, 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. Four *coocc* modules are synthesized in a single cell and combined in a grid of 5 times 12, for a total of 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,

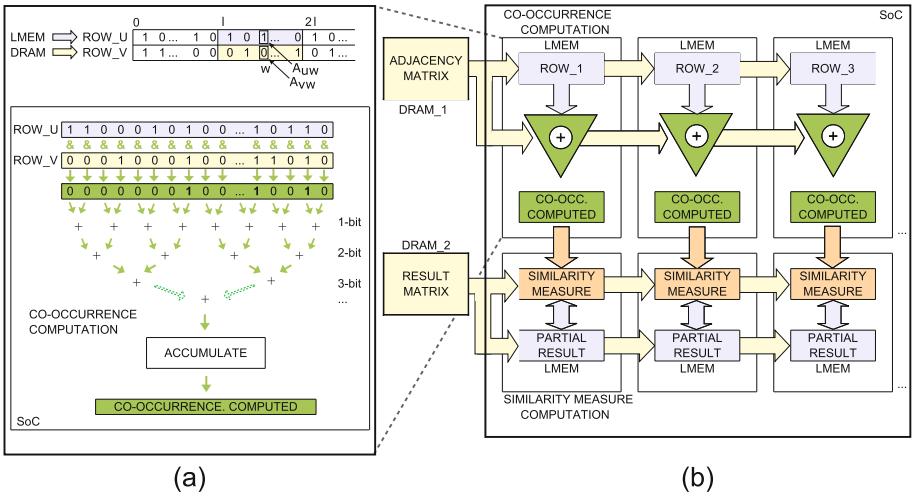

Fig. 4. The *coocc* and result module (b) works on one dataset after another, always updating the same result. It loads one row of the graph into the caches (local memory (LMEM)) and first calculates the *coocc* before calculating the similarity measures. The *coocc* module (a) consists of an efficient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

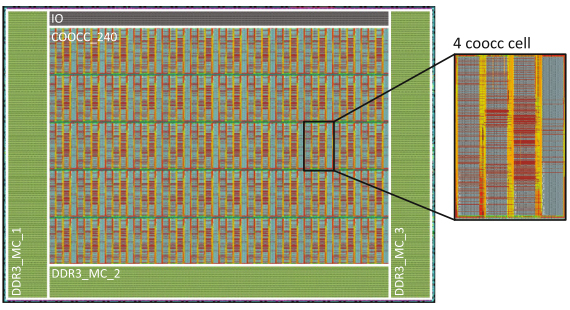

Fig. 5. ASIC layout in 28 nm technology. It consists of 240 *coocc* modules, three DRAM controllers (green) and IO logic. The swap randomization block is not visible here due to its small size .(Color figure online)

while for the results a shift register over the whole chip is used. That makes the architecture perfectly scalable.

In total, this architecture accumulates the *coocc* from $240 \times 256 = 61,440$ matrix columns per cycle, or $\sim 24.5 \times 10^{12}$ columns per second. In a comparison with the fastest CPU based population count [16] running at 3.4 GHz, that represents a speed up of $\sim 59\times$.

The rest of the design is occupied by memory controllers and IO, see Fig. 5. For the memory controllers, we have estimated the numbers based on the corresponding publications [6,10]. The whole ASIC has a size of 51.2 mm^2 and average power consumption of 11.7 W.

Partial-Line Cache Optimization. In a follow-up work [3 SPP], we further increased the efficiency of the hardware architecture by introducing the concept of partial line caches. Since the area of the *coocc* modules are dominated by their cache, reducing the cache size enables much higher degrees of parallelism. However, if the *coocc* modules cannot hold an entire row of the adjacency matrix, the partial results must be temporarily stored, raising the question of the optimal cache size for achieving the best performance.

As will be detailed in Sect. 3.4, higher granularity DRAM channels (shorter word-sizes) can be used to accelerate the graph randomization step. However, they increase the latency of accessing the adjacency matrix rows, therefore presenting the worst-case for the *coocc* computation.

In Fig. 6 we have simulated the time it takes to process the adjacency matrix with the time it takes to store the partial result on average for one line segment when using a channel word-size of 8-bit (the smallest possible). Since those operations are pipelined, the optimal cache size is given by the Pareto front between the two operations. The smallest latency is reached for a cache size of 8 kB.

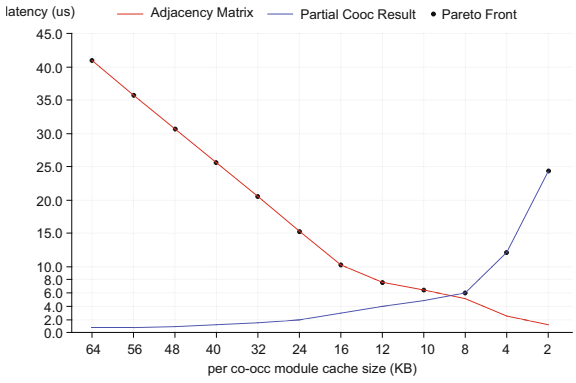

Fig. 6. Latencies of accessing the input data stored in Adjacency Matrix rows in comparison with latency for storing partial *coocc* results, assuming the same channel width for both memories involved in the design. The Pareto front is the maximum of each. Numbers are for 8-bit channel DRAMs.

While in the first design 240 *coocc* modules with 64 kB caches have been used, 8 kB caches allow us to increase the number of modules up to 1920 for the same total cache size. This results in a similar total chip area, from 51.2 mm^2 to 57.3 mm^2, as the caches dominate the *coocc* module in both cases. With this approach, we could further reduce the runtime of the *coocc* computation by a factor of 8× when using the same 64-bit channels, or maintain the same speed when using 8-bit channels.

3.4 Swap Randomization

With the accelerated *coocc* computation, the generation of each sample, i.e., the randomization of the graph becomes the bottleneck. Edge swapping is a strictly sequential operation in that any swap can depend on the result of the last swap, therefore its parallelization is not as straightforward as instantiating more processing units. Nevertheless, we addressed this bottleneck by exploiting the fine-grained access to DRAM [3 SPP], what is only possible when implementing our own memory controller, as well as a collision-aware swap parallelization.

Fine Grained DRAM Access. Most modern CPU have a fixed size interface of 64 bits with the DRAM. DRAM devices, however, can have higher granularity interfaces of, e.g., 8 bits (×8), and they are physically combined into groups of 8 devices to build the 64 bits interface. A fixed burst length of 8 DRAM accesses fills up one cache line of 512 bits, or 64 bytes. For any modern CPU, one cache line, i.e., 512 bits, is the minimum amount of data that can be loaded from DRAM.

Since the swap randomization operates only on single integers and single bits, reducing the word length of the DRAM interface increases the "computations per loaded bit" (the arithmetic intensity, see Fig. 3) immensely. Indirectly, of course, it also increases the performance because the swap randomization is bounded by the random memory access latency.

We have derived an alternative hardware architecture that slightly modifies the memory controller in order to address each of the DRAM devices (with 8 bit interfaces) independently [3 SPP]. Normally, the memory controller addresses all 8 DRAM devices of a memory channel as if it was a single device, i.e., it sends the same commands and addresses to all devices. This allows the memory channel to share the command and address lines for all devices, saving energy and area at the cost of having a common address space. The data lines (8 or 64 per ×8 or ×64 device), on the other hand, cannot be shared, as the data in each DRAM device must be transferred independently. By introducing a *chip select signal* and interleaving the commands to each DRAM device, we can transform the common address space into 8 independent ones. This works because, during the DRAM latency (data request to data ready), the DRAM device ignores address and command lines, as they get internally saved at the request moment. That way, we can load only $8 \times 8 = 64$ bits instead of $8 \times 64 = 512$ bits in one DRAM device access. This is a slight modification in the memory controller and channel, but one that could not be accomplished without custom hardware design.

For that scheme to be the most efficient, it requires that the data stored in each DRAM device to be independent. That is, each DRAM chip holds its own copy of the graph, as shown in Fig. 7(b). With that we can read or write 8 random numbers in the same time with a ×8 channel compared to a single with one ×64 channel, as shown in Fig. 7(c)(d). This scheme speeds up the swap randomization by a factor of 4× up to 8×.

Figure 7(a) shows the alternative architecture using two ×64 memory channels. This design is more suitable whenever the *coocc* calculation is the bottleneck of the algorithm, while the design in Fig. 7(b) provides faster graph randomization. This trade-off is depicted in Sect. 4.

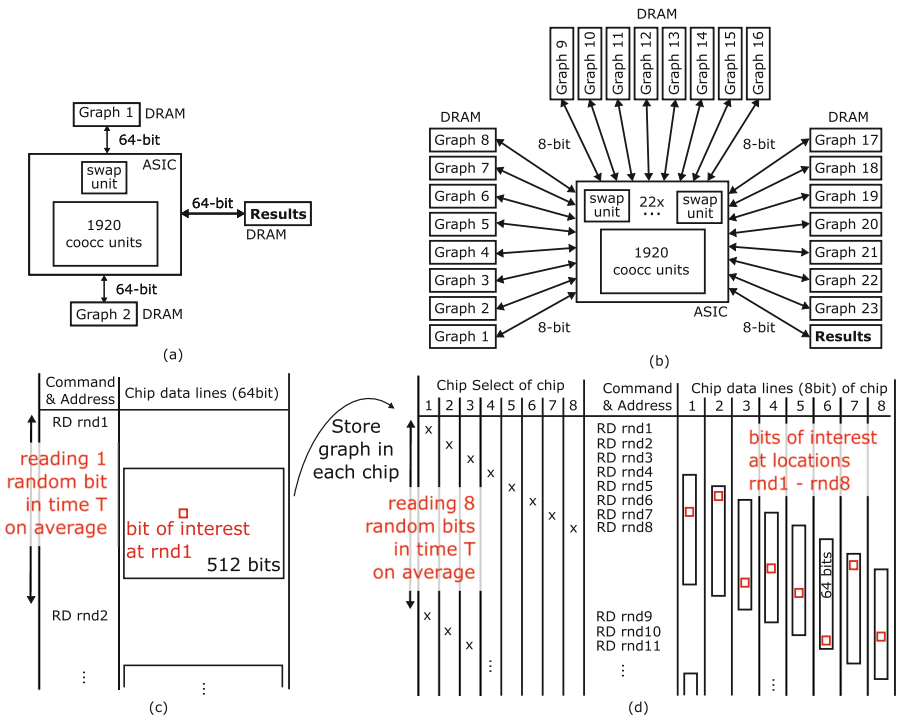

Fig. 7. Showing the ASIC for two memory configurations: ×64 (a) and ×8 (b) channels. In the case (a) only two graphs are stored and one swap unit is active, while in case (b) 23 graphs are stored and 22 swap units are active. Architecture (a) is useful for small number of swaps, while architecture (b) is useful for high number of swaps. Showing how the different random reads are performed for a ×64 channel (c) and ×8 channels (d). By interleaving the random accesses of 8 swap units with chip select over one command and address channel, 8 reads can be performed for (d) in the same time as one read for (c). This results in an 8× speedup.

Collision-Aware Swap Parallelization. Edge swapping is an inherently sequential operation in that every step can depend on the previous ones. For

large graphs with millions of edges, we access the memory at random locations for billions of chained swaps. Even then, we can divide the edge swapping chain into chunks that can be processed in parallel, if we make sure that none of the swaps depend on the previous ones in the same chunk. These chunks can be reordered by the memory controller in order to ensure the minimum amount of random accesses.

We have simulated the performance of the swap parallelization for different chunk sizes with the DRAMSys tool [12]. For that, we created trace files that describe the access pattern to the DRAM. The speedup saturates at 2.5× for a chunk size of $N = 12$ parallel swaps. Since N is small, checking for collisions between swaps is much faster than writing the swapped edges back to DRAM, therefore it does not incur any time overhead.

Table 2. Cluster ASIC Comparison

Implementation	Memory [GB]	Runtime [hour]	Power [W]	Energy [MJ]
Low number of swaps ($\|nodes\| \ln \|nodes\|$):				
ASIC (1920 modules, 64-bit channels)[a]	5.3	1.51	20.1	0.11
10 node Intel cluster[b]	202	8.5 (5.6x)	3700	114 (1030x)
ASIC (240 modules, 64-bit channels)[c]	4.6	9.0 (6.0x)	15.8	0.51 (4.6x)
High number of swaps ($\|edges\| \ln \|edges\|$):				
ASIC (1920 modules, 8-bit channels)[a]	30.9	11.1	13.3	0.53
10 node Intel cluster[b]	202	16 (1.4x)	3300	190 (360x)
ASIC (240 modules, 64-bit channels)[c]	4.6	483 (44x)	10.9	19 (36x)

[a]node including: ASIC with 1920 *coocc* modules, 28 nm; 48 GB DDR3 memory (×64 or ×8 channels); board (ethernet, clocks), power supply.
[b]each node: 2×Intel Xeon X5680 @ 12 × 3.33 GHz, 32 nm; 48 GB DDR3 memory
[c]node including: ASIC with 240 *coocc* modules, 28 nm; 8 GB DDR3 memory (×64 channel); board (ethernet, clocks), power supply.

4 Performance Comparison

For demonstrating the performance of our design we have calculated the similarity measures for the Netflix Prize data set[6], specifically the good ratings (4 or 5 stars) from users to movies. The resulting graph has 17,769 movies, 478,615 users, and 56,919,190 edges. In this case, V_l are the the movies, V_r the users.

In practice, the number of swaps in the randomization process is chosen between $|nodes| \ln |nodes| = 6{,}259{,}639$ and $|edges| \ln |edges| = 1{,}016{,}414{,}121$. To demonstrate that our design qualifies for the full range, we compare it for both of those extremes. The exact number in practice usually depends on the

[6] Available at https://www.kaggle.com/netflix-inc/netflix-prize-data. Last accessed on 24/11/2022.

nature of the graph. A heuristic for determining the optimal number of swaps is discussed in Chapter 3.

Table 2 compares our ASIC and our optimized cluster implementations of the LA algorithm. The cluster implementation was developed specifically for this reference work and tested on two Intel Xeon X5680 @ 12×3.33 GHz, 32 nm server nodes. Optimization involved the selection of an algorithm that minimizes computing time for the given memory resources, removing locks by data partitioning, and data access linearization [4 SPP,3 SPP].

Our first ASIC design (240 *coocc* modules) has a runtime performance comparable to the cluster implementation if a low number of swaps is necessary. Notice, however, that it becomes almost useless (takes 20 days to complete) if $|edges| \ln |edges|$ swaps are required. This is clear since in this first architecture we only focused on accelerating the *coocc* calculation. Still, the total energy consumption is 10x lower (notice that the total energy takes into account the total runtime). This goes to show the amount of energy overhead for software implementations, or how much energy can be saved by task specific ASICs. This conclusion is interesting for both ends of the computing spectrum: The embedded computing systems that are limited by battery capacity, size, and power constraint, and the high performance computing, limited by energy expenses and power dissipation issues.

Our second design shows how reconfigurability can address data-dependent bottlenecks (i.e., the *coocc* or the edge swapping). By using smaller word-sizes ($\times 8$ channels), we can accelerate both the *coocc* and edge swapping in such a way that the Link Assessment becomes 45% faster than the cluster implementation while consuming 360x less energy. When fewer swaps are necessary, the word-size can be increased to $\times 64$ channels, further reducing the *coocc* computation time (the primary bottleneck), reaching a speed up of 5.6x compared to software. The total energy economy, in this case, is even more impressive: From 114 MJ in software to only 0.11 MJ in the custom design. This is partially due to the large reduction of 38x in main memory footprint, from 202 GB to 5.3 GB.

5 Conclusion

Further increasing computational performance in modern technologies has become a key challenge for the whole hardware and software industry. Phenomena such as *Dark Silicon* force system designers to move to highly heterogeneous systems, consisting of a large amount of highly dedicated hardware accelerators in combination with classical programmable architectures such as CPUs and GPUs. Since hardware accelerators focus on specific tasks, they can be much more power/energy and compute efficiently than the latter ones.

In this chapter, we present a hardware architecture for the Link Assessment (LA) algorithm, used for cleaning up noise data in large graphs. Processing and analyzing large graphs will remain a key application in HPC for the next decades. Since the current bottleneck for speeding up this task is fast random access to memory, with standard DRAM architectures and controllers on commodity

HPC nodes we experience a hard performance limit, together with high energy consumption.

Our proposed architecture uses custom data structures and exploits bit-wise access to the data in order to overcome these limitations. On a 28 nm ASIC device with a DDR3 controller it is 1030x more energy efficient compared to a standard compute cluster, using 38x less memory in total. We show multiple optimization techniques that are specific to custom hardware designs, such as a slight memory controller modification that reduces the average random access latency; and a tailored cache design that enables scalable parallelism w.r.t. memory bandwidth. The architecture is fully flexible and can also be ported as an FPGA accelerator solution. This clearly illustrates the potential of hardware accelerators for the LA in particular and the graphs analysis domain in general.

Transferring the concepts to other algorithms such as Curveball (see Chapter 2) is the subject of ongoing work.

References

1. Asanovic, K., et al.: A view of the parallel computing landscape. Commun. ACM **52**(10), 56–67 (2009). https://doi.org/10.1145/1562764.1562783
2. Brugger, C.: A new approach to efficient heterogeneous computing = Ein neuer Ansatz für effiziente, heterogene Datenverarbeitung. Ph.D. thesis, University of Kaiserslautern, Germany (2016)

3 SPP. Brugger, C., Grigorovici, V., Jung, M., de Schryver, C., Weis, C., Wehn, N., Zweig, K.A.: A memory centric architecture of the link assessment algorithm in large graphs. IEEE Des. Test **35**(1), 7–15 (2018). https://doi.org/10.1109/MDAT.2017.2750900

4 SPP. Brugger, C., et al.: A custom computing system for finding similarties in complex networks. In: ISVLSI, pp. 262–267. IEEE Computer Society (2015). https://doi.org/10.1109/ISVLSI.2015.78

5. Duranton, M., et al.: Hipeac vision 2019. European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC) (2019)
6. Dutoit, D., et al.: A 0.9 pJ/bit, 12.8 GByte/s WideIO memory interface in a 3D-IC NoC-based MPSoC. In: Symposium, VLSIT, pp. C22–C23. IEEE (2013)
7. Esmaeilzadeh, H., Blem, E.R., Amant, R.S., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. IEEE Micro **32**(3), 122–134 (2012). https://doi.org/10.1109/MM.2012.17
8. Garraghan, P., Al-Anii, Y., Summers, J., Thompson, H., Kapur, N., Djemame, K.: A unified model for holistic power usage in cloud datacenter servers. In: UCC, pp. 11–19. ACM (2016). https://doi.org/10.1145/2996890.2996896
9. Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 197–208. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77220-0_21
10. Howard, J., et al.: A 48-core IA-32 message-passing processor with DVFS in 45 nm CMOS. In: ISSCC, pp. 108–109. IEEE (2010). https://doi.org/10.1109/ISSCC.2010.5434077

11. Jung, M.: System-level modeling, analysis and optimization of dram memories and controller architectures. Ph.D. thesis, University of Kaiserslautern, Germany (2017)
12. Jung, M., Weis, C., Wehn, N.: Dramsys: a flexible DRAM subsystem design space exploration framework. IPSJ Trans. Syst. LSI Des. Methodol. **8**, 63–74 (2015). https://doi.org/10.2197/ipsjtsldm.8.63
13. Lee, E.A., et al.: The swarm at the edge of the cloud. IEEE Des. Test **31**(3), 8–20 (2014). https://doi.org/10.1109/MDAT.2014.2314600
14. Miller, B.A., et al.: A scalable signal processing architecture for massive graph analysis. In: ICASSP, pp. 5329–5332. IEEE (2012). https://doi.org/10.1109/ICASSP.2012.6289124
15. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002). https://doi.org/10.1126/science.298.5594.824
16. Mula, W., Kurz, N., Lemire, D.: Faster population counts using AVX2 instructions. Comput. J. **61**(1), 111–120 (2018). https://doi.org/10.1093/comjnl/bxx046
17. de Schryver, C.: Design methodologies for hardware accelerated heterogeneous computing systems. Ph.D. thesis, University of Kaiserslautern, Germany (2014)
18. Slavakis, K., Giannakis, G.B., Mateos, G.: Modeling and optimization for big data analytics: (statistical) learning tools for our era of data deluge. IEEE Signal Process. Mag. **31**(5), 18–31 (2014). https://doi.org/10.1109/MSP.2014.2327238
19 SPP. Spitz, A., Gimmler, A., Stoeck, T., Zweig, K.A., Horvát, E.: Assessing low-intensity relationships in complex networks. PLoS ONE **11**(4), 1–17 (2016). https://doi.org/10.1371/journal.pone.0152536
20. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785
21 SPP. Zweig, K.A., Brugger, C., Grigorovici, V., De Schryver, C., Wehn, N.: Automated determination of network motifs (2015)
22 SPP. Zweig, K.A., Kaufmann, M.: A systematic approach to the one-mode projection of bipartite graphs. Soc. Netw. Analys. Min. **1**(3), 187–218 (2011). https://doi.org/10.1007/s13278-011-0021-0