Substream-Centric Maximum Matchings on FPGA

Maciej Besta, Marc Fischer, Tal Ben-Nun, Johannes De Fine Licht, Torsten Hoefler
Department of Computer Science, ETH Zurich

ABSTRACT

Developing high-performance and energy-efficient algorithms for maximum matchings is becoming increasingly important in social network analysis, computational sciences, scheduling, and others. In this work, we propose the first maximum matching algorithm designed for FPGAs; it is energy-efficient and has provable guarantees on accuracy, performance, and storage utilization. To achieve this, we forego popular graph processing paradigms, such as vertex-centric programming, that often entail large communication costs. Instead, we propose a *substream-centric* approach, in which the input stream of data is divided into substreams processed independently to enable more parallelism while lowering communication costs. We base our work on the *theory of streaming graph algorithms* and analyze 14 models and 28 algorithms. We use this analysis to provide theoretical underpinning that matches the physical constraints of FPGA platforms. Our algorithm delivers high performance (more than 4× speedup over tuned parallel CPU variants), low memory, high accuracy, and effective usage of FPGA resources. The substream-centric approach could easily be extended to other algorithms to offer low-power and high-performance graph processing on FPGAs.

ACM Reference Format:
Maciej Besta, Marc Fischer, Tal Ben-Nun, Johannes De Fine Licht, Torsten Hoefler, Department of Computer Science, ETH Zurich . 2019. Substream-Centric Maximum Matchings on FPGA. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293916

1 INTRODUCTION

Analyzing large graphs has become an important task. Example applications include investigating the structure of Internet links, analyzing relationships in social media, or capturing the behavior of proteins [2, 43]. There are various challenges related to the efficient processing of such graphs. One of the most prominent ones is the size of the graph datasets, reaching trillions of edges [13]. Another one is the fact that processing such graphs can be very power-hungry [4].

Deriving and approximating *maximum matchings* (MM) [9] are important and well-known graph problems. A matching in a graph is a set of edges that have no common vertices. Maximum matchings are used in computational sciences, image processing, VLSI design, or scheduling [9, 59]. For example, a matching of the carbon skeleton of an aromatic compound can be used to show the locations of double bonds in the chemical structure [59]. As deriving the exact MM is usually computationally expensive, significant focus has been placed on developing fast approximate solutions [17].

To enable high-performance graph processing, various schemes were proposed, such as vertex-centric approaches [24], streaming [54], and others [58]. These approaches have the advantage of being easily deployable in combination with the existing processing infrastructure such as Spark [62]. However, they were shown to be often inefficient [46] and they are not explicitly optimized for power-efficiency.

To enable power-efficient graph processing, several graph algorithms and paradigms for FPGAs were proposed [8, 18, 19, 23, 37, 48, 50, 61, 64–67]. Unfortunately, *none of them targets maximum matchings*. In addition, the established paradigms for designing graph algorithms that were ported to FPGAs, for example the vertex-centric paradigm, *are not straightforwardly applicable to the MM problem* [55].

In this work, we propose *the **first** design and implementation of approximating maximum matchings on FPGAs*. Our design is power-efficient *and* high-performance. For this, we forego the established vertex-centric paradigm that may result in complex MM codes [55]. Instead, basing on *streaming theory* [26], we propose a *substream-centric* FPGA design for deriving MM. In this approach, we ❶ divide the incoming stream of edges into *substreams*, ❷ process each substream independently, and ❸ merge these results to form the final algorithm outcome.

For highest power-efficiency, we execute phases ❶–❷ on the FPGA; both phases work in the streaming fashion and offer much parallelism, and we identify the FPGA as the best environment for these phases. Conversely, the final gathering phase, that usually takes $< 1\%$ of the total processing time as well as consumed power and exhibits little parallelism, is conducted on the CPU for highest performance.

To provide formal underpinning of our design and thus enable guarantees of correctness, memory usage, or performance, we base our work on the family of *streaming models* that were developed to tackle large graph sizes. A special case is the *semi-streaming model* [26], created specifically for graph processing. It assumes that the input is a sequence of edges (pairs of vertices), which can be accessed only sequentially in one direction, as a stream. The main memory (can be randomly accessed) is assumed to be of size $O(n \operatorname{polylog}(n))$[1] ($n$ is the number of vertices in the graph). Usually, only one pass over the input stream is allowed, but some algorithms assume a small (usually constant or logarithmic) number of passes. We investigate *a total of 14 streaming models* and *a total of 28 MM algorithms* created in these models, and use the insights from this investigation to develop our MM FPGA algorithm, ensuring both empirical speedups and provable guarantees on runtime, used memory, and correctness.

[1] $O(\operatorname{polylog}(n)) = O(\log^c(n))$ for some constant $c \in \mathbb{N}$

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02…$15.00
https://doi.org/10.1145/3289602.3293916

Towards these goals, we contribute:

- the first design and implementation of the maximum matching algorithm on FPGAs,
- an in-depth analysis of the potential of using streaming theory (14 models and 28 algorithms) for accelerating graph processing on FPGAs,
- a substream-centric paradigm that combines the advantages of semi-streaming theory and FPGA capabilities,
- detailed evaluation and high speedups over state-of-the-art baselines on both CPUs and FPGAs.

2 BACKGROUND AND NOTATION

We first present the necessary concepts.

2.1 Graph-Related Concepts

Graph Model We model an undirected graph G as a tuple (V, E); $V = \{v_1, ..., v_n\}$ is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. Vertex labels are $\{1, 2, ..., n\}$. If G is weighted, it is modeled by a tuple (V, E, w); $w(e)$ or $w(u, v)$ denote the weight of an edge $e = (u, v) \in E$. The maximum and minimum edge weight in G are denoted with w_{max} and w_{min}. G's adjacency matrix is denoted by A.

Compressed Sparse Row (CSR) In the well-known CSR format, A is represented with three arrays: *val*, *col*, and *row*. *val* contains all A's non-zeros (that correspond to G's edges) in the row major order. *col* contains the column index for each corresponding value in *val*. Finally, *row* contains starting indices in *val* (and *col*) of the beginning of each row in A. CSR is widely adopted for its simplicity and low memory footprint for sparse matrices.

Graph Matching A *matching* $M \subseteq E$ in a graph G is a set of edges that share no vertices. M is called *maximal* if it is no longer a matching once any edge not in M is added to it. M is *maximum* if there is no matching with more edges in it. Maximum matchings (MM) in unweighted graphs are called *maximum cardinality* matchings (MCM). Maximum matchings in weighted graphs are called *maximum weighted* matchings (MWM). Example matchings are illustrated in Figure 1.

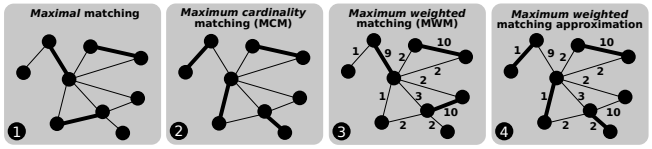

Figure 1: **Example matchings**. The edges in matchings are represented by bold lines, edge weights are represented with numbers.

Maximum Weighted Matching Given a weighted graph $G = (V, E, w)$, a maximum weighted matching is a matching M^*, such that its weight $w(M^*) = \sum_{e \in M^*} w(e)$ is maximized. An algorithm provides an ε-approximation of M^*, if – for any derived matching M – it holds that $w(M^*)/w(M) \leq \varepsilon$.

2.2 Architecture-Related Concepts

FPGAs FPGAs aim to combine the advantages of Application Specific Integrated Circuits (ASICs) and CPUs: they offer ASIC's high performance and low power usage, and they can be reconfigured to enable execution of arbitrary circuits. Usually, the FPGA clock frequency is ≈200MHz, dependent on the algorithm and the FPGA platform. This is an order of magnitude less compared to high-end CPUs (up to 4.7GHz [33]) and below GPUs (up to 1.5GHz [49]). However, due to the custom design deployed directly in hardware, multiple advantages such as low power consumption arise.

FPGA Components Configurable Logical Blocks (CLBs) [57], also known as Adaptive Logic Modules (ALMs) [60], implement the FPGA custom logic. To improve locality and reduce wiring overhead, CLBs are grouped together into clusters (called fabrics [57] or LABs [60]). Next, Block Random Access Memory (BRAM) allows to store small amounts of data (up to 20 kbits per BRAM [31]) and provides fast data access, acting similarly to a CPU cache. Usually, hundreds of BRAM units are distributed over a single FPGA.

FPGA+CPU Hybrid computation systems consist of a host CPU and an attached FPGA. First ❶, an FPGA can be added to the system as an accelerator; the host main memory is separated from the FPGA private DRAM memory and data must be transferred over PCIe. Often, the FPGA is configured as a PCIe endpoint with a direct memory access (DMA) controller, allowing to move data between the host and the FPGA without the need of CPU resources. PCIe is high-bandwidth oriented, but exhibits high overhead and latency for small packets [15]. This drawback is overcome by storing often accessed data in the private DRAM using the memory controller, or storing the data on chip in the FPGA's BRAM. Second ❷, the CPU and the FPGA can be directly linked by an interconnect, such as Intel's QuickPath Interconnect (QPI), providing a coherent view to a single shared main memory. Examples of these systems include Intel HARP [51] and the Xilinx Extensible Processing Platform [56]. The direct main memory access allows to share data without the need to copy it to the FPGA. To prevent direct physical main memory accesses, HARP provides a translation layer, allowing the FPGA to operate on virtual addresses. It is implemented in both hardware as a System Protocol Layer (SPL) and in software, for example as a part of the Centaur framework [52]. Moreover, a cache is available to reduce access time. According to Choi et al. [15], systems with direct interconnect exhibit lower latency and higher throughput than PCIe connected FPGAs. *In our substream-centric FPGA design for deriving MM, we use a hybrid CPU+FPGA system to take advantage of both the CPU and the FPGA in the context of graph processing.*

3 FROM SEMI-STREAMING TO FPGAS

We first summarize the analysis into the theory of streaming models and algorithms. We conducted the analysis to provide formal underpinning of our work and thus ***ensure provable properties, for example correctness, approximation, or performance***. Towards this goal, *we analyzed 14 different models of streaming* (simple streaming [30], semi-streaming [26], insert-only [26], dynamic [5], vertex-arrival [16], adjacency-list [45], cash-register [47], Turnstile [47], sliding window [20], annotated streaming [11], StreamSort [3], W-Stream [22], online [38], and MapReduce [21]) and *28 different MM algorithms*. We present the full analysis in a separate report[2]. Here, we only provide the final outcome: the best candidates for adoption in the FPGA setting are ❶ **semi-streaming** graph algorithms that ❷ **expose parallelism by decomposing the**

[2] https://spcl.inf.ethz.ch/Parallel_Programming/Matchings-FPGA

Reference	Approx.	Space	#Passes	Wgh[1]	Gen[2]	Par[3]
[26]	$1/2$	$O(n)$	1	👎	👍	👍
[41, Theorem 6]	$1/2+0.0071$	$O(n\ \text{polylog}(n))$	2	👎	👍	👍
[41, Theorem 2]	$1/2+0.003^*$	$O(n\ \text{polylog}(n))$	1	👎	👍	👍
[36, Theorem 1.1]	$O(\text{polylog}(n))$	$O(\text{polylog}(n))$	1	👎	👍	👍
[26, Theorem 1]	$2/3-\varepsilon$	$O(n \log n)$	$O\left(\log\left(1/\varepsilon\right)/\varepsilon\right)$	👎	👎	👍
[6, Theorem 19]	$1-\varepsilon$	$O\left(n\ \text{polylog}(n)/\varepsilon^2\right)$	$O\left(\log\log\left(1/\varepsilon\right)/\varepsilon^2\right)$	👎	👎	👍
[41, Theorem 5]	$1/2+0.019$	$O(n\ \text{polylog}(n))$	2	👎	👎	👍
[41, Theorem 1]	$1/2+0.005^*$	$O(n \log n)$	1	👎	👎	👍
[41, Theorem 4]	$1/2+0.0071^*$	$O(n\ \text{polylog}(n))$	2	👎	👎	👍
[39]	$1-1/e$	$O(n\ \text{polylog}(n))$	1	👎	👎	👍
[28, Theorem 20]	$1-1/e$	$O(n)$	1	👎	👎	👍
[35, Theorem 2]	$1-\frac{e^{-k}k^{k-1}}{(k-1)!}$	$O(n)$	k	👎	👎	👍
[14]	1	$\tilde{O}\left(k^2\right)$	1	👎	👍	👍
[14]	$1/\varepsilon$	$\tilde{O}\left(n^2/\varepsilon^3\right)$	1	👎	👍	👍
[7, Theorem 1]	n^ε	$\tilde{O}\left(n^{2-3\varepsilon}+n^{1-\varepsilon}\right)$	1	👎	👎	👍
[26, Theorem 2]	6	$O(n \log n)$	1	👍	👍	❓
[44, Theorem 3]	$2+\varepsilon$	$O(n\ \text{polylog}(n))$	$O(1)$	👍	👍	❓
[44, Theorem 3]	5.82	$O(n\ \text{polylog}(n))$	1	👍	👍	❓
[63]	5.58	$O(n\ \text{polylog}(n))$	1	👍	👍	❓
[25]	$4.911+\varepsilon$	$O(n\ \text{polylog}(n))$	1	👍	👍	❓
[29]	$3.5+\varepsilon$	$O(n\ \text{polylog}(n))$	1	👍	👍	❓
[53]	$2+\varepsilon$	$O\left(n \log^2 n\right)$	1	👍	👍	❓
[27]	$2+\varepsilon$	$O(n \log n)$	1	👍	👍	❓
[26, Section 3.2]	$2+\varepsilon$	$O(n \log n)$	$O\left(\log_{1+\varepsilon/3} n\right)$	👍	👍	❓
[6, Theorem 28]	$\frac{1}{1-\varepsilon}$	$O\left(n \log(n)/\varepsilon^4\right)$	$O\left(\varepsilon^{-4} \log n\right)$	👍	👍	👍
[6, Theorem 22]	$\frac{1}{\frac{2}{3}(1-\varepsilon)}$	$O\left(n\left(\frac{\varepsilon \log n-\log \varepsilon}{\varepsilon^2}\right)\right)$	$O\left(\varepsilon^{-2} \log\left(\varepsilon^{-1}\right)\right)$	👍	👍	👍
[6, Theorem 22]	$\frac{1}{1-\varepsilon}$	$O\left(n\left(\frac{\varepsilon \log n-\log \varepsilon}{\varepsilon^2}\right)\right)$	$O\left(\varepsilon^{-2} \log\left(\varepsilon^{-1}\right)\right)$	👍	👎	👍
[17]	$4+\varepsilon$	$O(n\ \text{polylog}(n))$	1	👍	👍	👍

Table 1: (§ 3) **Comparison of algorithms for maximum matching**. *Approximation in expectation, [1]**Wgh**: accepted weighted graphs, [2]**Gen**: accepted general (non-bipartite) graphs, [3]**Par**: Potential for parallelization; k is the size of a given maximum matching. 👍: A given feature is offered. 👎: A given feature is not offered. In the context of parallelization: 👍: a given algorithm is based on a method that is easily parallelizable (e.g., sampling), 👍: a given algorithm uses a method that may be complex to parallelize (e.g., augmenting paths), ❓: it is unclear how to parallelize a given algorithm (e.g., it is based on a greedy approach).

incoming stream of edges for independent processing, for example the MM algorithm by Crouch and Stubbs [17].

3.1 Why Semi-Streaming?

The *semi-streaming model* [26] was created specifically for graph processing. It assumes that processing the incoming stream of edges can utilize at most $O(n\ \text{polylog}(n))$ random memory. ***Thus, algorithms under this model may address the limited FPGA BRAM capacity better than algorithms in models with weaker memory-related constraints***.

3.2 Which Semi-Streaming MM Algorithm?

Table 1 compares the considered semi-streaming and related MM algorithms. We identify those with properties suggesting an effective and versatile FPGA design: low space consumption, one pass, and applicability to general graphs. Finally, virtually all designed algorithms are approximate. Yet, as we show later (§ 5), in practice they deliver near-accurate results.

We conjecture that the majority of the considered MM algorithms deliver limited performance on FPGA because *their design is strictly sequential*: every edge in the incoming stream can only be processed after processing the previous edge in the stream is completed. However, we identify some algorithms that introduce a certain amount of parallelism. Here, we focus on the algorithm by Crouch and Stubbs [17], used as a basis for our FPGA design (last row of Table 1). We first outline this algorithm and then justify our selection.

Algorithm Intuition The MWM algorithm by Crouch and Stubbs [17] delivers a $(4+\varepsilon)$-approximation of MWM. It consists of two parts. In Part 1, one selects L subsets of the incoming (streamed) edges and computes a *maximum cardinality* matching for each such subset. In Part 2, the derived maximum matchings are combined into the final *maximum weighted matching*. The approach is visualized in Figure 2.

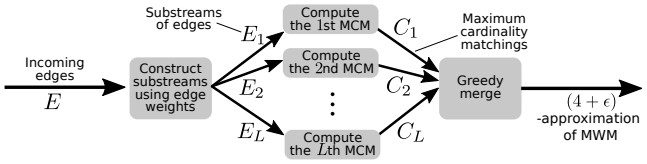

Figure 2: **The design of the MWM algorithm** of Crouch and Stubbs [17].

Algorithm Details The algorithm of Crouch and Stubbs [17] provides a $(4+\varepsilon)$-approximation to the MWM problem assuming an unordered stream of incoming edges with possible graph updates (edge insertions). The basic idea is to reduce the MWM problem to $L \equiv O(\text{polylog}(n))$ instances of the MCM problem. Given the input stream of incoming edges E, $O\left(\frac{1}{\varepsilon}\log n\right)$ many substreams are generated. Each substream E_i is created by filtering the edges according to their weight. Specifically, we have $E_i = \{e \in E \mid w(e) \geq (1+\varepsilon)^i\}$. Since an edge that belongs to substream $i+1$ also belongs to substream i, it holds that $E_{i+1} \subseteq E_i$. Next, for each substream, an MCM C_i is constructed. The final $(4+\varepsilon)$-approximation to MWM is greedily constructed by considering the edges of every C_i, in the descending order of i.

We select this algorithm as the basis of our substream-centric FPGA design because it ❶ can be straightforwardly parallelized, ❷ ensures only $O(n\ \text{polylog}n)$ memory footprint as it belongs to the semi-streaming model, ❸ targets general weighted graphs, ❹ its structure matches well the design of a hybrid FPGA+CPU system: while substreams can be processed in parallel on the FPGA, the greedy sequential merging of substreams into the final MWM can be done on the CPU, and ❺ it requires only one pass over the streamed dataset of size $O(m+n)$, limiting expensive data transfers between the FPGA and DRAM. Now, we could not find other algorithms that would clearly satisfy all the above criteria simultaneously. However, we do not conclude that other algorithms are unsuitable for an efficient FPGA implementation, and leave developing such designs as future work.

3.3 How To Adapt Semi-Streaming to FPGAs?

In § 4, we describe the FPGA adaptation, design, and implementation of the selected semi-streaming MM algorithm. We stream the edges as stored in the CSR representation. Our substream-centric design implements a staged pipeline with throughput of up to one edge per cycle.

4 MAXIMUM MATCHING ON FPGA

We now describe the design and implementation of the substream-centric maximum matching for FPGAs.

4.1 Overview of the Algorithm

We start with a high-level overview of the MWM algorithm. A pseudo code is shown in Listing 1. For each edge, we iterate in the descending order of i over the L substreams, identifying them by their respective weights (Line 11). The

```
1  //Input: ε, E, L. Output: T (a (4+ε)-approximation of MWM).   I/O
2
3  //PART 1 (Stream processing): compute L maximum matchings
4  C: List of Lists; //L lists to store edges in L substreams
5  MB: Matrix; //The matching bits matrix of size L×n
6  substream_weights: List; //The list of substream weights;
7  //substream_weights[i] = (1+ε)^i.
8  has_added: bool; //Controlling adding an edge to only one MCM
9  foreach(WeightedEdge e : E) {
10   has_added = false;
11   for(i = L-1; i >= 0; i--) {
12     if(e.weight >= substream_weights[i]) {
13       if(!MB[e.u][i] && !MB[e.v][i]) {
14         MB[e.u][i] = 1; MB[e.v][i] = 1;
15         if(!has_added) {//Add e only once to the matchings
16           C[i].add(e); has_added = true;
17 } } } } }                                                  FPGA
18
19 //PART 2 (Post processing): combine L matchings into a MWM
20 T: List; //A list with the edges of the final MWM
21 tbits: List; //An array containing the matching bits of T
22 for(i = L-1; i >= 0; i--) {
23   foreach(WeightedEdge e : C[i]) {
24     if(!tbits[e.u] && !tbits[e.v]) {
25       tbits[e.u] = 1; tbits[e.v] = 1;
26       T.add(e);
27 } } }
28 return T;                                                  CPU
```

Listing 1: (§ 4.1) **The high-level overview of the substream-centric MWM**, based on the scheme by Crouch and Stubbs [17]

```
1  //Input and Output: as in Listing 1.                       I/O
2
3  //PART 1 (Stream processing): compute L maximum matchings
4  for(Epoch k = 1; k <= ⌈n/K⌉; k++) {
5    Load u-matching bits from DRAM into double-buffered BRAM
6    Merge the K rows of edges (loaded from DRAM into one stream S)
7      with a merging network (Figure 4), apply lexicographic order
8    //Process each edge
9    foreach(WeightedEdge e : S) {
10     Matching bits requester loads matching bits (e.v) from DRAM
11     //Apply the 8 stage pipeline for each edge
12     Stage 1: extract v-matching bits from a data chunk,
13       determine BRAM address
14     Stage 2: load the matching bits for e.u from BRAM
15     Stage 3: wait for one cycle due to the latency of the BRAM
16     Stage 4: store the arriving BRAM data in a register, select
17       the correct matching bits, compute el[i] = e.w = (1+ε)^i
18     Stage 5: compute the matching
19     Stage 6: write u-matching bits to BRAM, write
20       v-matching bits to double-buffered BRAM if required
21     Stage 7: determine the least significant bit in te,
22       store them in variable i
23     Stage 8: write the edge to DRAM at C[i]
24       (if part of a matching), write v-matching bits to DRAM
25   }
26   Wait till all writes to DRAM are committed}               FPGA
27
28 //PART 2 (Post processing): As in Listing 1.               CPU
```

Listing 2: (§ 4.1–§ 4.4) The pseudocode of the substream-centric MWM algorithm, **enhanced with the blocking optimization and a lexicographic ordering**.

i-th substream weight is given by $(1+\varepsilon)^i$. For each maximum matching C_i, we use a bit matrix MB to track if a vertex has an incident edge to *ensure that C_i remains a matching* (i.e., that no two vertices share an edge). Bits included in MB are called *matching bits*. Bits in MB associated with a vertex u, *the source vertex of a processed edge*, (u-matching bits) determine if u has an incident edge included in some matching; they are included in column mb_u of matrix MB. Matching bits associated with vertex v, *the destination vertex of a processed edge*, (v-matching bits) track the incident edges of v; they are included in column mb_v of matrix MB. Since there are L matchings and n vertices, the bit matrix MB is a matrix of size $L \times n$. Furthermore, every matching stores its edges in a list. If an edge is added, a flag is set to `true` to prevent that the edge is added to multiple lists (Line 16). This reduces the runtime of the post-processing part, in which we iterate in the descending order over the L lists of edges to generate the $(4+\varepsilon)$-approximation to the maximum weighted matching.

Time & Space Complexity The space complexity is $O(nL)$ to track the matching bits, and $O(\min(m, n/2)L\log(n))$ to store the edges of L maximum matchings. The time complexity is $O(mL)$ for substream processing on the FPGA and $O(nL)$ for substream merging on the CPU, giving $O(mL+nL)$.

Reducing Data Transfer with Matching Bits Storage We assume that the input is streamed according to the CSR order corresponding to the input adjacency matrix. If we process a matrix row, we load the edges from DRAM to the FPGA. Further, we can store the matching bits mb_u of vertex u in BRAM on the FPGA, *since they are reused multiple times*. The matching bits of v are streamed in from DRAM. Since the matching bits for v are not used afterwards for the same matrix row, we write them back to DRAM. Using this approach, we can process the whole graph row by row and need to store only the u-matching bits in BRAM.

4.2 Blocking Design for More Performance

Problem of Data Dependency We cannot start processing the next row of the adjacency matrix until the last matching bits of the previous row have been written to DRAM, because we might require accessing the same v-matching bits again (read after write dependency). In such a design, the waiting time required after each row could grow, decreasing performance.

Solution with Blocking Rows We alleviate the data dependency by applying *blocking*. We merge K adjacent rows to become one stream; we call the merged stream of K rows an *epoch*, and denote the k-th epoch (starting counting from 1) as k. There are $\lceil n/K \rceil$ epochs in total. To enable merging the rows, we define a *lexicographic ordering* over all edges.

Lexicographic Ordering Let a tuple (u, v, w, k) denote an edge with vertices u, v, weight w, and associated epoch $k = \lfloor (u-1)/K \rfloor + 1$. Then, the lexicographic ordering is given by: $(u_a, v_a, w_a, k_a) < (u_b, v_b, w_b, k_b)$ iff $k_a < k_b \vee (k_a = k_b \wedge v_a < v_b) \vee (k_a = k_b \wedge v_a = v_b \wedge u_a < u_b)$; the edge weight is ignored. An example is in Figure 3 (top). The lexicographic ordering is implemented by a simple merging network.

Advantages of Blocking At the end of each epoch, v-matching bits are written to DRAM. This reduces the number of such transfers from n to n/K. Moreover, if edges in different rows share the same v-matching bits, only one load from DRAM is required. Finally, u-matching bits can be kept in BRAM, since they are reused multiple times.

Further Optimizations To achieve a performance of *up to one processed edge per cycle*, we pipeline the processed edges, we distribute the u-matching bits over multiple BRAMs (to facilitate reading data from different addresses), and we double buffer u-matching bits to reduce latencies.

4.3 Input and Output Format

The input to the FPGA algorithm is a custom variant of the Compressed Sparse Row (CSR) format. An example is given in Figure 3 (bottom). The format has two parts: The `pointer_data` and the `graph_data`. First, the `pointer_data` stores information about the start and end of each row of the adjacency matrix. An entry contains three parts: the ID of the *data chunk* with information about where the first edge is stored, the data chunk offset denoting the offset of the first edge from the start of the data chunk, and the number of associated edges (a data chunk refers to data of a given size at

an aligned memory address). Each entry uses 32 bits, making an entry of the `pointer_data` 96 bits. We fit five entries (480 bits) in a data chunk. Second, the `graph_data` is a stream of edges. One entry consists of the column index and the edge weight. The row identifier is given by the corresponding entry in the `pointer_data`. One `graph_data` entry requires 64 bits, allowing to store eight edges in a data chunk.

Our custom data layout has different advantages over the usual CSR format. First, a single entry of the `pointer_data` already gives all required information about the start and length of the row of the adjacency matrix. This entails some redundancy compared to the traditional CSR, but only requires one load from DRAM to resolve a given edge. Further, CSR splits the column indices and values. We merge them together in one stream, reducing the number of random accesses.

The output of the FPGA consists of L substreams of edges. The i-th stream contains edges of the maximum matching C_i. We use 128 bits for each edge: 32 bits each for the vertex IDs, the edge weight, and the assigned index i of the maximum matching (which could be omitted). A single data chunk therefore contains four output edges.

4.4 Details of Processing Substreams on FPGA

We explain the interaction of the FPGA modules dedicated to generating the lexicographic ordering (Part 1) and computing the maximum matchings (Part 2); see Figure 4 and Listing 2.

4.4.1 Generating Lexicographic Ordering. As input to the FPGA, we get the address pointing to the start of `pointer_data`, the number of vertices n, the number of edges m, a pointer p_{out} where we write the output to, and an offset value o to distinguish the L output streams (start of output stream i is at $p_{out} + i \cdot o$). The **pointer requester** is responsible for requesting the data chunks holding the `pointer_data`. The requested pointers arrive at the **pointer receiver**. Given a data chunk, the pointer receiver unwraps the five pointers, and passes them to the **edge requester**. The `pointer_data` from the pointer receiver is passed to four different queues Q_0, Q_1, Q_2, Q_3, where

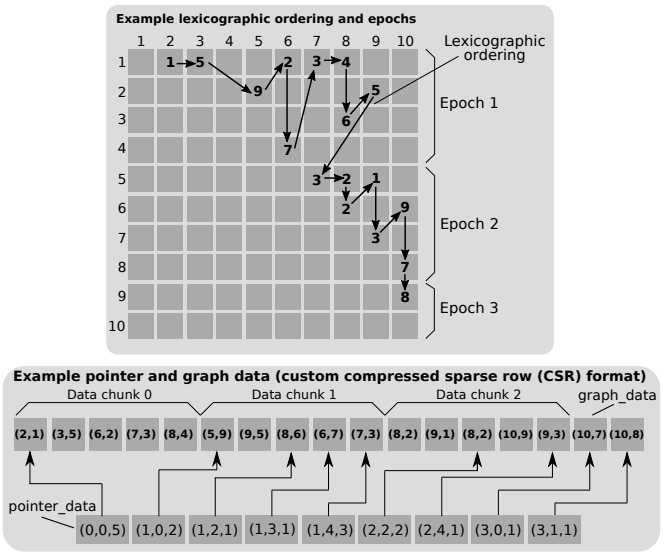

Figure 3: An example input adjacency matrix, its annotated lexicographic ordering illustrated by arrows ($K = 4$), and and its custom compressed sparse row (CSR) format. The entries of the adjacency matrix denote the weight of an edge.

every queue gets a subset of the pointers dependent on K. Assume for simplicity that the vertex IDs start at 0 and $(K \mod 4) = 0$. Then, to be precise, given a pointer $p(u)$ pointing to row u, we assign $p(u)$ to Q_i if $(u \mod K) \geq K/4 \cdot i \wedge (u \mod K) < K/4 \cdot (i+1)$. For example, with $K = 16$, Q_0 stores $p(u)$ with $u = 0, 1, 2, 3, 16, 17, 18, 19, 31 \ldots$, and Q_1 stores $p(u)$ with $u = 4, 5, 6, 7, 20, \ldots$. The `pointer_data` is loaded from the queues into a BRAM array BP of size K, where every entry holds two pointers ($2K$ pointers are therefore stored in total). If an entry i of BP has pointers $p(u')$ and $p(u'')$, it holds that $i = (u' \mod K) = (u'' \mod K)$ and $p(u'')$ requests edges for an epoch after $p(u')$. Therefore, only the first pointer in an entry is valid to use and we have random access to K valid pointers in total. To describe the mechanism that determines the selection of the next pointer to request new edges, we first inspect further processing steps.

The **edge receiver** gets data chunks containing `graph_data` from the framework and unwraps them (we use the Centaur framework [52] to access main memory independently of the CPU). Information regarding the offset and number of edges which are valid for a data chunk request is also passed from the edge requester to the edge receiver. Next, an edge $e = (u, v, w)$ is passed from the edge receiver to the **merger**. There, the edge is inserted in a *starting queue* (with ID $(u \mod K)$). The merger merges the K streams in lexicographic ordering. It consists of a series of merging elements, where each element has two input queues and an output port. The element compares edges in its queues and outputs the edges according to the lexicographic ordering. The merging elements form a binary tree, such that for a given K, there are $K/2$ starting elements with K starting queues in total.

The edge requester can observe the size of the starting queues of the merger. It operates in two modes to determine a pointer to new edges. In mode 1, one selects a pointer $p(u)$ from queue Q_i as the next candidate if the corresponding starting (merger) queue $(u \mod K)$ does not overflow, and store the pointer in BRAM BP at position $(u \mod K)$. If mode 1 fails (for example, if there is no empty space at the appropriate position in BP), then mode 2 selects the pointer according to the merger starting queue which has the least amount of edges. Note that the edge requester also takes the requests which are in flight into account to predict the future size of the starting queue. This approach ensures that the merger queues do not overflow and their load is balanced.

For a row u which has no edges, a special information is passed from the edge requester to the edge receiver. It then inserts an artificial edge in the merger. This allows to overcome problems, where a merging element waits for new input, but does not receive any, since the adjacency matrix row is empty. The merging network filters these edges at the output port (they are not passed on).

4.4.2 Deriving L Maximum Matchings. The stream in lexicographic ordering is passed to the **matching bits requester**. This module requests the v-matching bits from DRAM. It can only operate when the bits of the epoch before have been acknowledged. Also, it only processes edges belonging to the current epoch which is defined by the **state controller**. The requested data is received in the **matching bits receiver**.

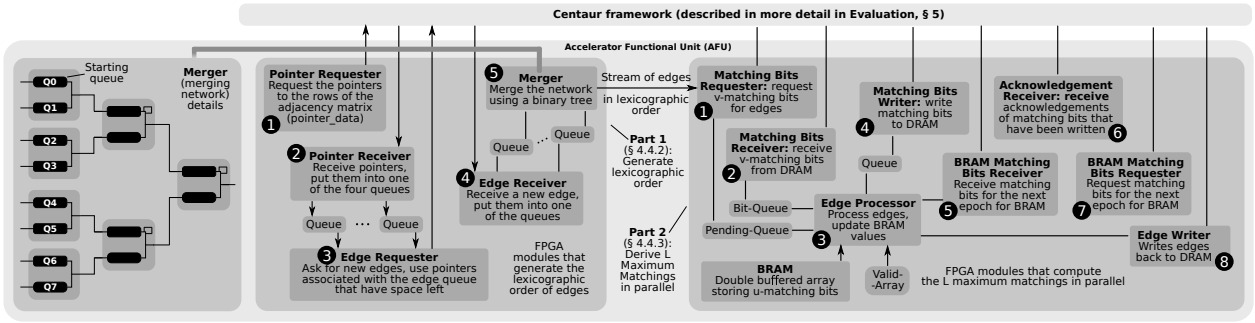

Figure 4: (§ 4.4) The **interaction of the FPGA modules** to approximate MWM. For clarity, the State Controller and the Read/Write Interface modules are omitted.

It passes the full data chunk to the **edge processor**. Using the matching bits and the ordered stream of edges, the edge processor computes the L maximum matchings in parallel in an 8-stage pipeline (Listing 2, Lines 10–24). In Stage 1, v-matching bits for a given edge are extracted from a data chunk. Further, the address of the u-matching bits in BRAM is computed. Since the more up-to-date v-matching bits might also be stored in BRAM, this address is also determined. In Stage 2, read requests to fetch the matching bits from BRAM are issued. Stage 3 only waits one clock cycle for BRAM to return the data. In Stage 4, the BRAM data arrives and is stored in a register. The stage also decides if v-matching bits are taken from the data chunk or from BRAM. Further, the stage computes the matching value *te* indicating if an edge $e = (u, v, w)$ belongs to substream E_i; $te[i] = w \geq (1+\varepsilon)^i$ for $i \in \{0, \ldots, L-1\}$. In Stage 5, the actual matching is computed. As the BRAM data from Stage 4 may already be obsolete, the computed values are also stored in registers for instant access in the next cycle. The result is passed to Stage 6, in which the updated u-matching bits (and if required also the v-matching bits) are written back to BRAM. In Stage 7, the maximum matching with the highest index, to which the edge is assigned, is determined. Finally, Stage 8 passes the edge to the **edge writer** to write it back to DRAM (if the edge is used in a matching) and also passes the updated v-matching bits to the **matching bits writer** for writing back to DRAM.

The BRAM storing the u-matching bits is double buffered. While the first BRAM buffer is used in the edge processor, the matching bits for the next epoch are loaded from DRAM to the second BRAM buffer. Since an epoch can alter the u-matching bits required for the next epoch, we write the according updates also in the double buffered BRAM if required. To prevent that stale data from DRAM overwrites the more up-to-date data, we use a register (the valid-array) as flag. After an epoch, the access is redirected to the BRAM containing the loaded data. The **BRAM matching bits requester** requests the according data from DRAM, and the **BRAM matching bits receiver** unwraps the data chunks. It passes the data to the edge processor. There, Stage 6 checks for data from the BRAM matching bits receiver and updates the according entry in the BRAM.

The **acknowledgement receiver** tracks the number of write acknowledgements from the framework and determines if all v-matching bits have been committed to DRAM when an epoch ends. When all edges from the epoch are processed, the state controller indicates the start of the next epoch.

4.5 Substream Merging on the CPU

After the L MCMs are written to DRAM, the CPU inspects them in the decreasing order to compute the final maximum matching $(4+\varepsilon)$-approximation.

5 EVALUATION

We now illustrate the advantages of our hybrid (CPU+FPGA) MWM design and inspect resource and energy consumption. *For every benchmark, each tested algorithm was synthesized, routed, and executed on the hybrid FPGA platform specified below.*

Compared Algorithms Since to our best knowledge *no MWM algorithms for FPGAs are available*, we compare our design to three state-of-the-art CPU implementations. In total, we evaluate three CPU and two CPU+FPGA algorithms; see Table 2. First ❶, we implement a sequential CPU-only version of the substream-centric MWM, based on the scheme by Crouch and Stubbs [17], as presented in Listing 1 (CS-SEQ). Second ❷, we parallelize the algorithm with OpenMP's `parallel-for` statement to compute different maximum matchings in parallel (CS-PAR). Third ❸, we implement the algorithm by Ghaffari [27] (G-SEQ) that provides a $(2+\varepsilon)$-approximation to MWM with time complexity of $O(m)$ and space complexity of $O(n\log(n))$ bits. Thus, this algorithm is optimal in the asymptotic time and space complexity. We compare these three algorithms to our optimized FPGA+CPU implementation, SC-OPT ❹ (SC-SIMPLE ❺ is consistently outperformed by SC-OPT and we thus usually exclude it for clarity of presentation). *To our best knowledge,* ***we report the first performance data for deriving maximum matchings on the FPGA***.

Implementation Details We implement our algorithms on a hybrid CPU+FPGA system using the Centaur framework [52], which provides a standard interface to the Accelerator Functional Unit (AFU), the custom FPGA implementation, allowing to access main memory independently of the CPU. Centaur consists of a software and a hardware part. The software part allows to start and stop hardware functions, to allocate and deallocate the shared memory, and pass input parameters to the FPGA. The hardware part is responsible for bootstrapping the FPGA, setting up the QPI endpoint, and handling reads and writes to the main memory.

Algorithm	Platform	Time complexity
Crouch et al. [17] Sequential (CS-SEQ)	CPU	$O(mL+nL)$
Crouch et al. [17] Parallel (CS-PAR)	CPU	$O(mL/T+nL)$
Ghaffari [27] Sequential (G-SEQ)	CPU	$O(m)$
Substream-Centric, no blocking (SC-SIMPLE)	Hybrid	$O(m+nL^2)$
Substream-Centric, with blocking (SC-OPT)	Hybrid	$O(m+n/K+nL)$

Table 2: (§ 5) Overview of **the evaluated MWM algorithm implementations**.

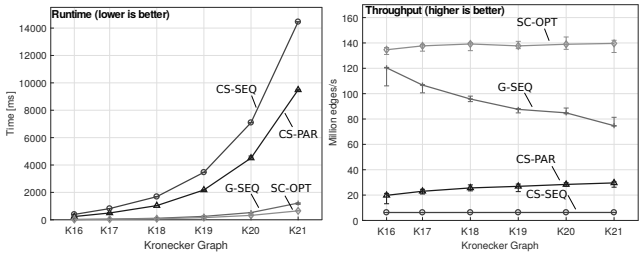

Figure 5: (§ 5.1) **Influence of graph size n on performance** (synthetic power-law graphs). $K = 32, L = 64, T = 4, \varepsilon = 0.1$.

Setup We use Intel HARP 2 [51], a hybrid CPU+FPGA system. It is a dual socket platform where one socket is occupied by an Intel Broadwell Xeon E5-2680 v4 CPU [32] with 14 cores (28 threads) with up to 3.3 GHz clock frequency. Each core has 32 KByte L1 cache and there is 35 MByte L3 cache in total. An Arria-10 FPGA is in the other socket. The used FPGA has speed grade 2 [34]. It provides 55 Mbit in 2,713 BRAM units and 427,200 ALMs. The FPGA is connected to the CPU by one QPI and two PCIe links. The system runs Ubuntu 16.04.3 LTS with kernel 4.4.0-96 as the operating system. All host code is compiled with gcc 5.4.0 and the -O3 compile flag.

Datasets The input graphs are shown in Table 3. We use both synthetic (Kronecker) power-law graphs of size up to $n = 2^{21}, m = 48n$ from the 10th DIMACS challenge [1] and real world KONECT [40] and SNAP [42] graphs. For unweighted graphs, we assigned weights uniformly at random with a fixed seed. The value range is given by $[1, (1+\varepsilon)^{L-1}+1]$.

Graph	Type	Reference	m	n
Kronecker	Synthetic power-law	DIMACS 10 [1]	$\approx 48n$	$2^k; k = 16, \ldots, 21$
Gowalla	Social network	KONECT [40]	950,327	196,591
Flickr	Social network	KONECT [40]	33,140,017	2,302,925
LiveJournal1	Social network	SNAP [42]	68,993,773	4,847,571
Orkut	Social network	KONECT [40]	117,184,899	3,072,441
Stanford	Hyperlink graph	KONECT [40]	2,312,497	281,903
Berkeley	Hyperlink graph	KONECT [40]	7,600,595	685,230
arXiv hep-th	Citation graph	KONECT [40]	352,807	27,770

Table 3: Selected used graph datasets. Kx denotes a Kronecker graph with 2^x vertices.

Measurements The runtime is measured by `clock_gettime` with parameter `CLOCK_MONOTONIC_RAW`, allowing the nanosecond resolution. The runtime of the FPGA implementations is determined by the Centaur framework. We execute each benchmark ten times to gather statistics and we use box plot entries to visualize data distributions.

5.1 Scaling Size of Synthetic Graphs

We first evaluate the impact from varying graph sizes (synthetic power-law Kronecker graphs), for the fixed amount of parallelism (the *weak scaling* experiment). The results are illustrated in Figure 5. The throughput for CS-SEQ and CS-PAR stays approximately constant below ≈12M edges/s. G-SEQ decreases in performance as the graph size increases. We conjecture that this is due to the increasing size of the hash map used to track pointers. This increases the time for inserts and deletes, and might also require re-allocations to increase the space. The performance for SC-OPT increases from ≈135M to ≈140M edges/s. This is because the initial (constant) overhead (due to reading from DRAM) becomes less significant with larger graphs. **We conclude that the substream-centric SC-OPT outperforms comparison targets for all consider sizes of power-law Kronecker graphs.**

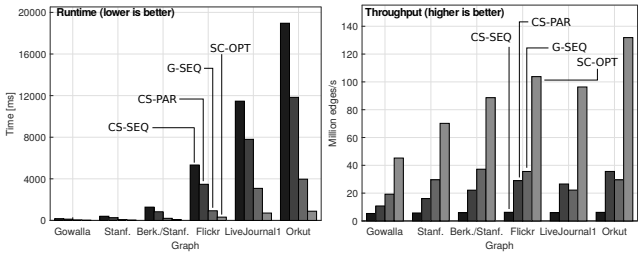

Figure 6: (§ 5.2) **Influence of graph dataset G on performance** (real-world graphs). $K = 32, L = 64, T = 4, \varepsilon = 0.1$.

5.2 Processing Different Real-World Graphs

We next analyze the performance of the considered designs for different real-world graphs; the results are illustrated in Figure 6. CS-SEQ and CS-PAR achieve sustained ≈3M edges/s and ≈10M edges/s, respectively. The performance of SC-OPT is ≈45M edges/s for small graphs due to the initial overhead of reading data from DRAM. Compared to the experiment with Kronecker graphs, the performance of both SC-OPT and G-SEQ is lower for all graphs except Orkut. The reason is the average vertex degree: it equals ≈48 in Kronecker graphs compared to ≈14 in Flickr and LiveJournal1. If the ratio is high, G-SEQ can drop many edges without further processing in an early phase. This reduces expensive updates to the hash map and lists. For SC-OPT, the waiting time (of data dependencies) lowers the performance. Still, **substream-centric SC-OPT ensures highest performance for all considered real-world graphs**.

5.3 Scaling Number of Threads T

In the CPU versions, one can compute in parallel different maximum matchings in SC-PAR using T threads. In the following, we run a *strong scaling* experiment (fixed graph size, varying T) for a power-law Kronecker graph. Figure 7 illustrates the results. Since G-SEQ and CS-SEQ are not multithreaded, they do not scale with T. The parallelized CS-PAR reaches up to ≈40M edges/s, a ≈6× improvement over the sequential version, and an ≈14× improvement over the parallel version with one thread. Therefore, the algorithm is still ≈3× slower than SC-OPT which achieves up to ≈140M edges/s on the K20 Kronecker graph. Scaling is limited since the parallel version takes L passes over the stream, whereas the other CPU algorithms process the input in one pass. The bandwidth usage of the parallel version with $T = 64$ threads is ≈32 GB/s (≈44M edges and 64 passes in one second), assuming no data sharing. Note that we only parallelize the stream-processing part which computes the $L = 64$ maximum matchings. However, as our analysis shows that the post-processing part takes <1% of the computation time of the maximum matching, parallelization of post-processing would provide hardly any benefit. We conjecture that the scaling of SC-PAR stops due to bandwidth limitations and the limited computational resources of 14 cores. Finally, **SC-OPT is the fastest regardless of T used by other schemes**.

5.4 Approximation Analysis

We briefly analyze how well in practice SC-OPT approximates the exact MWM. The results are in Figure 8 (SC-OPT, SC-SIMPLE, CS-SEQ, and CS-PAR produce the same results).

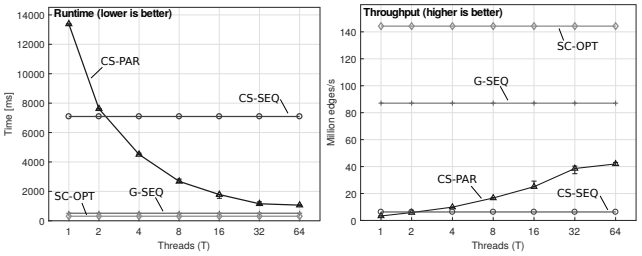

Figure 7: (§ 5.3) **Influence of the number of threads T on performance**. The input graph is Kronecker with $n = 2^{20}, K = 32, L = 64, \varepsilon = 0.1$.

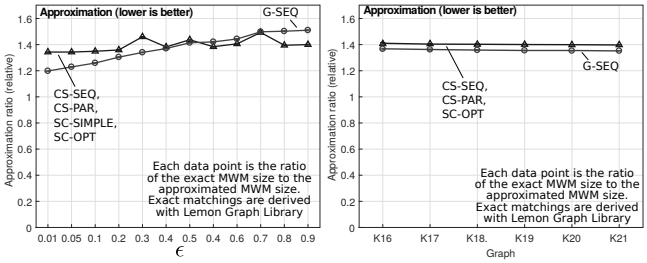

Figure 8: (§ 5.4) **Approximation analysis**. The input graph is Kronecker with $n = 2^{19}$ (left). $L = 128$, $T = 4$, and $\varepsilon = 0.1$ (right).

The accuracy is negligibly (≈3%) lower than that of G-SEQ for a fixed ε and varying n (Kronecker graphs). The higher ε becomes, the more advantage over G-SEQ SC-OPT has. As higher ε entails less circuit complexity (fewer substreams are processed independently, assuming a fixed L [17]), **we conclude that the substream-centric MWM SC-OPT scheme provides better approximation than G-SEQ when physical resources become more constrained.**

5.5 Influence of Blocking Parameter K

We also analyze the performance impact from K, a parameter that determines how many rows in the streamed-in adjacency matrix are merged together using a lexicographic ordering. Figure 9 illustrates the results. On one hand, the CPU schemes cannot take significant advantage when K increases, showing that no cache locality is exploited. On the other hand, FPGA-based SC-OPT accelerates from ≈125M to ≈175M edges/s. This is up to 2× faster than the work-optimal G-SEQ and up to 55× faster than CS-SEQ. This is expected as the amount of stalling is reduced by a factor of n/K. Moreover, increasing K allows to share more matching bits between edges. The performance impact is reduced when K reaches 256. We conjecture this is because of the random access to the matching bits, approaching the peak random bandwidth. Furthermore, G-SEQ outperforms all other CPU implementations with up to ≈90M edges/s. Compared to CS-SEQ (≈3.15M edges/s) and CS-PAR (≈5.6M edges/s), this is >15×. Finally, parallelization comes with high overhead, such that the four threads in CS-PAR achieve less than 2× speedup compared to CS-SEQ. **We conclude that our blocking scheme enables SC-OPT to achieve even higher speedups.**

5.6 Influence of Maximum Matching Count L

Finally, we analyze the impact of L on performance. L is the number of substreams and thus maximum matchings computed independently. CS-SEQ and CS-PAR achieve high performance with up to ≈400M edges/s for $L = 1$. The

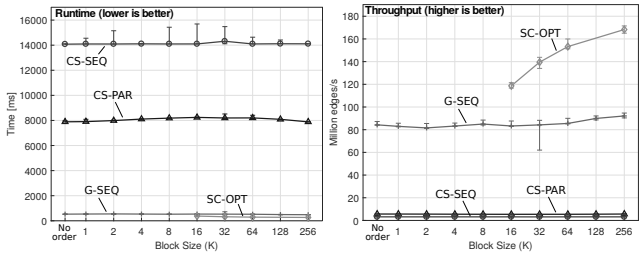

Figure 9: (§ 5.5) **Influence of epoch size K on the performance**. The input graph is Kronecker with $n = 2^{20}$. $L = 128$, $T = 4$, and $\varepsilon = 0.1$.

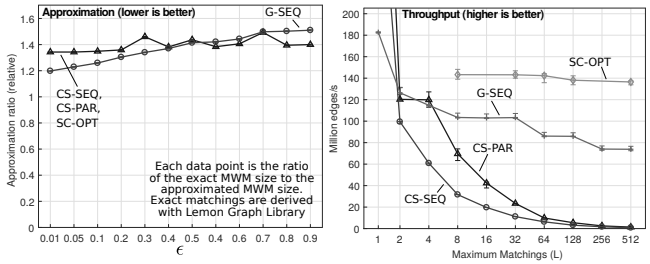

Figure 10: (§ 5.6) **Influence of L on performance**. The input graph is Kronecker ($n = 2^{20}, K = 32, T = 4$). As L changes, ε changes as follows: for $1 \leq L \leq 32$, we select $\varepsilon = 0.6$, for $64 \leq L \leq 128$ we select $\varepsilon = 0.1$, and for $256 \leq L \leq 512$ we select $\varepsilon = 0.03$; w_{max} is given by $w_{max} = (1+\varepsilon)^L$. We restricted the range of L for SC-OPT due to the significant runtime required to generate different bitstreams for evaluation.

performance drops linearly with L (X-axis has a logarithmic scale) to ≈800k edges/s for CS-SEQ and ≈1.3M edges/s for CS-PAR. G-SEQ also drops in performance as L increases due to ε and w_{max}. Since L increases, we also increase the range of the weight (L influences the approximation by $\varepsilon = \sqrt[L]{w_{max}} - 1$). Thus, for $L = 1$ the maximum edge weight is given by $w_{max} = 1$, allowing G-SEQ to drop many edges in an early phase. The drop of performance between $L = 32$ and $L = 64$ are due to a change in ε, requiring G-SEQ to store more data. Similarly, we change ε between $L = 128$ and $L = 256$. **SC-OPT keeps its performance at ≈140M edges/s (≈330ms) and outperforms other schemes.**

5.7 FPGA Resource Utilization

Table 4 shows the usage of FPGA resources. As maximum matchings are computed on the FPGA in one clock cycle, the number of computed matchings L influences the amount of used logic. Moreover, for SC-OPT, K and L determine the FPGA layout. Specifically, K influences the BRAM usage, since every element in the merging network requires two queues which are each mapped to one BRAM unit. We also consider the amount B [bits] of BRAM allocated to storing the matching bits. SC-OPT requires only 21% of Arria-10's BRAM and 32% out of all ALMs for a design that outperforms other targets by at least ≈2× (Figures 9–10); these speedups can be increased even further by maximizing circuitry utilization.

FPGA Algorithm	Parameters	Used BRAM	Used ALMs
SC-SIMPLE	$\log B = 12, L = 8$	5.6 MBit (10%)	89,388 (21%)
SC-SIMPLE	$\log B = 18, L = 6$	21 MBit (38%)	88,920 (21%)
SC-OPT	$K = 32, L = 512$	11.5 MBit (21%)	151,998 (32%)
SC-OPT	$K = 256, L = 128$	24.8 MBit (45%)	350,556 (82%)

Table 4: (§ 5.7) **FPGA resource usage** for different parameters.

5.8 Energy Consumption

We estimate the energy consumption of SC-SIMPLE and SC-OPT using the Altera PowerPlay Power Analyzer Tool; see

Table 5. Furthermore, the host CPU (Broadwell Xeon E5-2680 v4) has TDP of 120 Watt [32] when all cores are in use. This is an upper bound for CS-PAR at $T = 64$. **FPGA designs reduce consumed energy by *at least* ≈88% compared to the CPU.**

Algorithm	Parameters	Energy Consumption [W]
SC-SIMPLE	$\log B = 18, L = 6$	14.714
SC-SIMPLE	$\log B = 12, L = 8$	14.598
SC-OPT	$K = 32, L = 512$	14.789
SC-OPT	$K = 256, L = 128$	14.789
SC-OPT	$K = 32, L = 64$	14.657
CS-PAR	$T = 64$	120

Table 5: (§ 5.8) **Estimated energy consumption for different parameters**.

5.9 Design Space Exploration

We now briefly analyze the interaction between the performance of our FPGA design and the limitations due to the clock frequency. The resource usage, determined by L and B, influences the frequency upper bound due to wiring and logic complexity. We applied a grid search to derive feasible frequencies for SC-SIMPLE; see Figure 11. Dark grey indicates 400MHz, light grey indicates 200MHz. Two factors have shown to limit the performance. First ❶, while computing the matching, we use an addition with a variable that uses L bits. Thus, the addition complexity grows linearly with L. More importantly ❷, the BRAM signal propagation limits the frequency. For example, for SC-SIMPLE and $\log B = 13$, the place and route report shows that the reset signal to set all BRAM units to zero becomes the critical path.

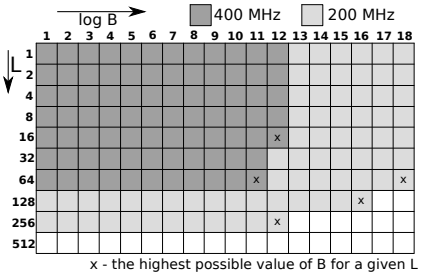

Figure 11: (§ 5.9) **Design space exploration: the used (available) frequencies**.

5.10 Optimality Analysis

We also discuss how far the obtained results are from the maximum achievable performance numbers; we focus on the most optimized SC-OPT. SC-OPT can process up to ≈175M edges/s. This is close to the optimum due to different reasons: Firstly, the implementation can process up to 1 edge per cycle (200M edges/s). Thus, the achieved performance is optimal within only ≈12%. Second, assuming that edges are read aligned from memory, it allows to read 8 edges per read request. Further, if every edge requires its own data chunk with matching bits, it needs 1 request per edge. Overall, this results in 1.25 read requests per edge. Under this assumption, the performance is limited to 160M edges/s. SC-OPT results are *higher* than this bound, which is possible because *the matching bits can be shared between edges*.

6 BEYOND SUBSTREAM-CENTRIC MM

We now briefly discuss how to apply our substream-centric FPGA design to other streaming graph algorithms. First, we identify some MM schemes that also divide the streamed dataset into substreams and can straightforwardly be adopted to the hybrid CPU+FPGA system. The **MWM algorithm by Grigorescu et al. [29]** reduces the MWM problem to $O(\varepsilon^{-1}\log(n))$ instances of maximum matchings, which could be processed on the FPGA analogously to our design; its merging phase could also be executed on the CPU. All our optimizations, such as blocking, are applicable in this case. Moreover, the **MWM algorithm by Feigenbaum et al. [26, Algorithm 4]** does not divide the stream of edges into substreams, but its design would potentially allow for applying our blocking scheme. A key part of this algorithm is maintaining a certain value q_e associated with each edge e. Given an edge $e = (u, v, w)$, q_e depends on values q_u and q_v associated with vertices u and v. We can apply the blocking pattern by storing q_u for u in BRAM, and streaming in q_v for v. Next, the **MWM algorithm by Ghaffari [27]** provides a $(2+\varepsilon)$-approximation. The algorithm compares the weights of incoming edges to values φ, indexed by u or v. Therefore, it can be computed on the FPGA using the blocking pattern by storing the values φ_u in BRAM, and streaming φ_v from DRAM, similarly to matching bits in our design. Further, as the algorithm requires postprocessing to derive the final result, it could be also delegated to the CPU.

We also identify algorithms unrelated to matching that could be enhanced with our design. The random **triangle counting algorithm by Buriol et al. [10]** is also a suitable candidate for the presented blocking pattern. The algorithm requires three passes. In pass 1, the number of paths of length two in the input graph is computed. In pass 2, a random path of length two is selected. In pass 3, the stream is searched for a certain edge, dependent on the randomly selected path. To reduce variance, passes 2–3 are run in parallel using a predetermined number of random variables (up to a million). This also implies that in pass 3 every edge in the stream must be checked against a million edges. To reduce the workload, a hash map used. The map is filled with edges which are expected to occur. We propose the following approach to exploit the blocking pattern: the CPU fills a hash map for each epoch with edges expected to arrive. The map is passed to the FPGA. The edges for this epoch are streamed in and compared to the pre-filled hash map. If the epoch changes, the next hash map is passed over.

7 RELATED WORK

Our work touches on various areas. We now discuss related works, briefly summarizing the ones covered in previous sections (streaming models in § 3 and streaming maximum matching algorithms in § 3.2, Table 1, and § 6).

Graph Processing on FPGAs The FPGA community has recently gained interest in processing graphs on FPGAs. First, some established CPU-related schemes were ported to the FPGA setting, for example vertex-centric [23, 24], GAS [65], edge-centric [67], BSP [37], and MapReduce [64]. There are also efforts independent of the above, such as FPGP [18], ForeGraph [19], and others [8, 37, 48, 50, 61, 66]. These works target popular graph algorithms such as BFS or PageRank. *None of them proposes any scheme for the important problem of finding graph matchings, targeted in this work.*

Graph Matchings and FPGAs The only work related to matchings and FPGAs that we are aware of merely uses matchings

to enhance FPGA segmentation design [12], *which is unrelated to deriving matchings and graph processing in general*.

Streaming Models and Algorithms We investigate the rich theory of streaming models [3, 5, 11, 16, 20–22, 26, 30, 38, 45, 47] and identify the semi-streaming model [26] as the best candidate for using together with FPGAs to deliver algorithms with provable properties that match FPGA characteristics such as limited memory. We then investigate semi-streaming algorithms for maximum matchings [6, 7, 14, 17, 25–29, 35, 36, 39, 41, 44, 53, 63] and identify the scheme by Crouch and Stubbs [17] that we use as the basis for our substream-centric design *that ensures low-power, high-performance, and high-accuracy general maximum weighted matchings on FPGAs*.

Hybrid FPGA+CPU Platforms Finally, our work is related to the study of hybrid CPU+FPGA platforms [52, 64]. We illustrate a case study with maximum matchings and show that *hybrid platforms can outperform state-of-the-art parallel CPU designs in both performance and power consumption*.

8 CONCLUSION

An important problem in today's graph processing is developing high-performance and energy-efficient algorithms for approximating maximum matchings. Towards this goal, we propose the first maximum matching algorithm for FPGAs. Our algorithm is *substream-centric*: the input stream is divided into substreams that are processed independently on the FPGA and merged into the final outcome on the CPU. This exposes parallelism while keeping communication costs low: only $O(m)$ data must be streamed from DRAM to the FPGA. Our algorithm is energy-efficient (88% less consumed energy over a tuned CPU variant) and provably accurate, fast (speedups of $>4\times$ over parallel CPU baselines), and memory-efficient ($O(nlog^c n)$ required storage).

The underlying FPGA design uses several novel optimizations, such as merging rows of the graph adjacency matrix and ordering resulting blocks lexicographically. This enables low utilization of FPGA resources (only 21% of Arria-10's BRAM and 32% out of all ALMs) while outperforming CPU baselines by at least $\approx 2\times$. Both the FPGA implementation and the substream-centric approach could be extended to other graph problems.

Finally, to the best of our knowledge, the proposed design is the first to combine the theory of streaming with the FPGA setting. Our insights coming from the analysis of 14 streaming models and 28 streaming matching algorithms can be used to develop more efficient FPGA designs.

Acknowledgements We thank Mohsen Ghaffari for inspiring discussions that helped us better understand graph streaming theory. We also thank David Sidler for his help with the FPGA infrastructure. Funded by the European Research Council (ERC) under the European Union's Horizon 2020 programme grant No. 678880. TBN is supported by the ETH Zurich Postdoctoral Fellowship and Marie Curie Actions for People COFUND program.

REFERENCES

[1] 10th DIMACS Challenge. Kronecker Generator Graphs, 2011.
[2] C. Aggarwal et al. Evolutionary network analysis: A survey. *CSUR*, 2014.
[3] G. Aggarwal et al. On the streaming model augmented with a sorting primitive. In *FOCS*, 2004.
[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. *Computer Architecture News*, 2016.
[5] K. J. Ahn et al. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, 2012.
[6] K. J. Ahn and S. Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. In *ICALP*, 2011.
[7] S. Assadi et al. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *SODA*, 2016.
[8] B. Betkaoui et al. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *FPT*, 2011.
[9] J. A. Bondy et al. *Graph theory with applications*. 1976.
[10] L. S. Buriol et al. Counting triangles in data streams. In *PODS*, 2006.
[11] A. Chakrabarti et al. Annotations in data streams. In *ICALP*, 2009.
[12] Y.-W. Chang et al. Graph matching-based algorithms for FPGA segmentation design. In *ICCAD*, 1998.
[13] A. Ching et al. One trillion edges: Graph processing at Facebook-scale. *VLDB*, 2015.
[14] R. Chitnis et al. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *SODA*, 2016.
[15] Y.-k. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *DAC*, 2016.
[16] G. Cormode et al. Independent sets in vertex-arrival streams. *arXiv:1807.08331*, 2018.
[17] M. Crouch and D. M. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *LIPIcs-Leibniz Inf.*, 2014.
[18] G. Dai et al. FPGP: Graph Processing Framework on FPGA. In *FPGA*, 2016.
[19] G. Dai et al. ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture. In *FPGA*, 2017.
[20] M. Datar et al. Maintaining stream statistics over sliding windows. *SIAM J. on Comp.*, 2002.
[21] J. Dean et al. MapReduce: simplified data processing on large clusters. *CACM*, 2008.
[22] C. Demetrescu et al. Trading off space for passes in graph streaming problems. *TALG*, 2009.
[23] N. Engelhardt and H. K.-H. So. Gravf: A vertex-centric distributed graph processing framework on FPGAs. In *FPL*, 2016.
[24] N. Engelhardt and H. K.-H. So. Vertex-centric Graph Processing on FPGA. In *FCCM*, 2016.
[25] L. Epstein et al. Improved approximation guarantees for weighted matching in the semi-streaming model. *J. on Discrete Mathematics*, 2011.
[26] J. Feigenbaum et al. On graph problems in a semi-streaming model. *Theoretical CS*, 2005.
[27] M. Ghaffari. Space-optimal semi-streaming for $(2+\varepsilon)$-approximate matching. *arXiv:1701.03730*, 2017.
[28] A. Goel et al. On the communication and streaming complexity of maximum bipartite matching. In *SODA*, 2012.
[29] E. Grigorescu et al. Streaming weighted matchings: Optimal meets greedy. *arXiv:1608.01487*, 2016.
[30] M. R. Henzinger et al. Computing on data streams. *External Mem. Alg.*, 1998.
[31] Intel. Intel Core i7-8700K Processor, 2017.
[32] Intel. Intel Xeon Processor E5-2680 v4, 2017.
[33] Intel. Stratix 10 GX/SX Device Overview, 2017.
[34] Intel Arria. Intel Arria 10 Device Overview, 2017.
[35] M. Kapralov. Better bounds for matchings in the streaming model. In *SODA*, 2013.
[36] M. Kapralov et al. Approximating matching size from random streams. In *SODA*, 2014.
[37] N. Kapre. Custom FPGA-based soft-processors for sparse graph acceleration. In *ASAP*, 2015.
[38] C. Karande et al. Online bipartite matching with unknown distributions. In *STOC*, 2011.
[39] R. M. Karp et al. An optimal algorithm for on-line bipartite matching. In *STOC*, 1990.
[40] KONECT. Konect network dataset, 2017.
[41] C. Konrad et al. Maximum matching in semi-streaming with few passes. *APPROX-RANDOM*, 2012.
[42] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, 2014.
[43] A. Lumsdaine et al. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 2007.
[44] A. McGregor. Finding graph matchings in data streams. In *APPROX-RANDOM*, 2005.
[45] A. McGregor et al. Better algorithms for counting triangles in data streams. In *PODS*, 2016.
[46] F. McSherry et al. Scalability! but at what COST? In *HotOS*, 2015.
[47] S. Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 2005.
[48] E. Nurvitadhi et al. Graphgen: An FPGA framework for vertex-centric graph computation. In *FCCM*, 2014.
[49] NVidia. GEFORCE GTX 1080 Ti, 2017.
[50] T. Oguntebi et al. Graphops: A dataflow library for graph analytics acceleration. In *FPGA*, 2016.
[51] N. Oliver et al. A reconfigurable computing system based on a cache-coherent fabric. In *ReConFig*, 2011.
[52] M. Owaida et al. Centaur: A framework for hybrid CPU-FPGA databases. In *FCCM*, 2017.
[53] A. Paz and G. Schwartzman. A (2+epsilon)-approximation for maximum weight matching in the semi-streaming model. In *SODA*, 2017.
[54] A. Roy et al. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*, 2013.
[55] S. Salihoglu et al. Optimizing graph algorithms on Pregel-like systems. *VLDB*, 2014.
[56] M. Santarini. Zynq-7000 EPP sets stage for new era of innovations. *Xcell*, 2011.
[57] L. Shang et al. Dynamic power consumption in Virtex™-II FPGA family. In *FPGA*, 2002.
[58] Y. Simmhan et al. Goffish: A sub-graph centric framework for large-scale graph analytics. In *EuroPar*, 2014.
[59] N. Trinajstić et al. On some solved and unsolved problems of chemical graph theory. *International Journal of Quantum Chemistry*, 1986.
[60] J. Tyhach et al. Arria™ 10 device architecture. In *CICC*, 2015.
[61] G. Weisz et al. Graphgen for coram: Graph computation on FPGAs. In *CARL*, 2013.
[62] M. Zaharia et al. Apache Spark: a unified engine for big data processing. *CACM*, 2016.
[63] M. Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.
[64] J. Zhang et al. Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search. In *FPGA*, 2017.
[65] J. Zhou et al. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *CCGRID*, 2017.
[66] S. Zhou et al. High-throughput and energy-efficient graph processing on FPGA. *FCCM*, 2016.
[67] S. Zhou et al. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. *SBAC-PAD*, 2017.