Engineering Motif Search for Large Motifs

Petteri Kaski
Department of Computer Science, Aalto University
Espoo, Finland
petteri.kaski@aalto.fi

Juho Lauri
Nokia Bell Labs
Dublin, Ireland
juho.lauri@nokia-bell-labs.com

Suhas Thejaswi
Department of Computer Science, Aalto University
Espoo, Finland
suhas.muniyappa@aalto.fi

Abstract

Given a vertex-colored graph H and a multiset M of colors as input, the *graph motif* problem asks us to decide whether H has a connected induced subgraph whose multiset of colors agrees with M. The graph motif problem is NP-complete but known to admit randomized algorithms based on *constrained multilinear sieving* over $\mathrm{GF}(2^b)$ that run in time $O(2^k k^2 m \mathrm{M}(2^b))$ and with a false-negative probability of at most $k/2^{b-1}$ for a connected m-edge input and a motif of size k. On modern CPU microarchitectures such algorithms have practical edge-linear scalability to inputs with billions of edges for small motif sizes, as demonstrated by Björklund, Kaski, Kowalik, and Lauri [ALENEX'15]. This scalability to large graphs prompts the dual question whether it is possible to scale to large motif sizes.

We present a *vertex-localized* variant of the constrained multilinear sieve that enables us to obtain, in time $O(2^k k^2 m \mathrm{M}(2^b))$ and for every vertex simultaneously, whether the vertex participates in at least one match with the motif, with a per-vertex probability of at most $k/2^{b-1}$ for a false negative. Furthermore, the algorithm is easily vector-parallelizable for up to 2^k threads, and parallelizable for up to $2^k n$ threads, where n is the number of vertices in H. Here $\mathrm{M}(2^b)$ is the time complexity to multiply in $\mathrm{GF}(2^b)$.

We demonstrate with an open-source implementation that our variant of constrained multilinear sieving can be engineered for vector-parallel microarchitectures to yield hardware utilization that is bound by the available memory bandwidth.

Our main engineering contributions are (a) a version of the recurrence for tightly labeled arborescences that can be executed as a sequence of memory-and-arithmetic coalescent parallel workloads on multiple GPUs, and (b) a bit-sliced low-level implementation for arithmetic in characteristic 2 to support (a).

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Mathematics of computing → Probabilistic algorithms, Theory of computation → Parallel algorithms

Keywords and phrases algorithm engineering, constrained multilinear sieving, graph motif problem, multi-GPU, vector-parallel, vertex-localization

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.28

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement 338077 "Theory and Practice of Advanced Search and Enumeration".

Acknowledgements We gratefully acknowledge the use of computational resources provided by the Aalto Science-IT project at Aalto University and by CSC – IT Center for Science, Finland.

1 Introduction

Computer microarchitectures are increasingly *vector-parallel*, or, *single-instruction-multiple-data* (SIMD) parallel. While such parallelism can be harnessed to an impressive effect in many applications, a number of applications still remain where vector-parallel algorithm designs have not yet been deployed in a manner that seeks to utilize the maximum parallel bandwidth obtainable, especially in terms of memory bandwidth. Problems with (sparse) graph inputs are a particular case where high-bandwidth vectorization is not immediate, in particular because traversing the edges of a graph via adjacency lists produces a data-dependent[1] pattern of memory accesses that does not vectorize easily, unless the algorithm design is such that each traversal of an edge involves accesses to one or more long vectors of consecutive words in memory. The protagonist of this paper is the following NP-complete graph problem, which we will show admits such vectorizable algorithm designs.

The Graph Motif Problem. The *graph motif* problem asks, given a vertex-colored graph H (the *host* graph) and a multiset M of colors (the *motif* or the *query*) as input, whether H has a connected induced subgraph whose multiset of colors agrees with M. The set of vertices of such a connected induced subgraph is called a *match* to the query. The graph motif problem appears, for example, as a query problem for protein interaction networks in computational biology (cf. Lacroix, Fernandes, and Sagot [24] and Bruckner, Hüffner, Karp, Shamir, and Sharan [8]); we postpone a further discussion of earlier work to the end of this section.

Asymptotically the currently fastest algorithm designs for the graph motif problem are based on algebraic methods. For a connected host graph with m edges and a motif of size k, *constrained multilinear sieving* (cf. Björklund, Kaski, and Kowalik [6]) over the finite field $\mathrm{GF}(2^b)$ enables a randomized algorithm that runs in time $O\big(2^k k^2 m \mathrm{M}(2^b)\big)$ and reports a false negative with probability at most $k/2^{b-1}$ (cf. Björklund, Kaski, Kowalik, and Lauri [7]). Here $\mathrm{M}(2^b) = O(b \log b)$ is the time complexity of multiplication in $\mathrm{GF}(2^b)$, cf. Lin, Al-Naffouri, Han, and Chung [25].

We observe in particular that the time complexity $O\big(2^k k^2 m \mathrm{M}(2^b)\big)$ of the Björklund–Kaski–Kowalik–Lauri design scales linearly in the number of edges m. Conversely, it is known that the exponential 2^k scaling in the motif size is the best possible unless there is a breakthrough in the complexity of the set cover problem (cf. Björklund, Kaski, and Kowalik [6, Theorem 6]).[2]

From a practical engineering perspective, constrained multilinear sieving is known to scale essentially in an edge-linear manner on modern CPU microarchitectures to graphs with hundreds of millions to billions of edges on a single compute node, when the motif size is small (cf. Björklund, Kaski, Kowalik, and Lauri [7]).

The present paper studies the dual question, namely what kind of empirical scalability can one obtain as a function of increasing motif size k. This question is motivated, for

[1] Indeed, while an adjacency list itself is read linearly from consecutive memory addresses, an algorithm typically must use the outcomes of such reads to address its further memory accesses, for example, to an array with one entry for each vertex in the graph.

[2] More precisely, an $O^*((2-\epsilon)^k)$-time design for some constant $\epsilon > 0$ for the graph motif problem would imply a $O^*((2-\delta)^n)$-time design for the set cover problem on a universe of size n for some constant $\delta > 0$.

example, when one is searching for a group of vertices with a specific color composition, but these vertices need not be immediately adjacent but rather some uncertain distance away from each other in the host graph. In this setting, the task can still be formulated as an instance of the graph motif problem,[3] but this instance will have a somewhat larger size k for the motif. In this situation, theory tells us that it will be difficult to obtain a base algorithm design with better than 2^k scalability in k, so one is essentially forced to seek efficiency through implementation engineering. Here constrained multilinear sieving is an excellent base design for vector-parallelization, in particular since the algorithm evaluates the same multivariate polynomial P (defined by H) at 2^k distinct points (defined by the colors of H and M together with randomization). Furthermore, one can use a small field size 2^b and yet obtain good control on the probability of false negatives.

Our Contributions. Using the open-source CPU-based parallel implementation of Björklund, Kaski, Kowalik, and Lauri [7] as a starting point, we engineer an implementation of motif search that runs on compute nodes with one or more GPUs with performance that for large motif sizes achieves the empirical peak transfer bandwidth of the GPU on-device memory. Our present implementation is tailored for NVIDIA GPUs, but relies on design principles that generalize to other vectorized microarchitectures, including CPUs with vector units.

In more detail, our contributions are as follows:

(i) *Vertex-localized sieving.* We develop a novel variant of the constrained multilinear sieve that operates simultaneously on a family of multivariate polynomials, one polynomial P_i for each vertex $i \in V(H)$, rather than a single polynomial P as in the original design. This re-design comes at asymptotically no extra cost and it enables us to *localize the outcome at vertices*, that is, with a single run of the sieve, we obtain *for every vertex* i the output whether there is at least one match that contains i, with a per-vertex probability of a false negative of at most $k/2^{b-1}$. This localization is advantageous in situations when the motif is large and the host has only isolated[4] matches to the motif.

(ii) *Coalescent recurrence for tightly labeled arborescences.* The most performance-critical aspect of the constrained multilinear sieve is the recurrence for the 2^k evaluations of each of the n polynomials P_i. We engineer a version of the recurrence that can be executed on multiple GPUs as a sequence of k workloads. For large k, these workloads are arithmetic-and-memory coalescent to the length of the available vectorization. A key principle is to design the memory layout of the recurrence so that each thread can work with the widest coalesced per-thread load and store instructions supported by the architecture. This is (i) to saturate the memory pipeline, and (ii) to supply each thread with enough local work that can be executed in low-latency per-thread registers to enable effective latency hiding. (Cf. Volkov [35] and Mei and Chu [27] for a discussion of latency hiding and memory hierarchies on GPUs.)

(iii) *Bit-sliced arithmetic in characteristic 2.* Constrained multilinear sieving runs over a finite field $\mathrm{GF}(2^b)$, which requires a fast implementation for finite-field multiplication. Asymptotically it is known that one can multiply in time $\mathrm{M}(2^b) = O(b \log b)$, cf. Lin,

[3] Potentially with generalization to each vertex being colored with a set of colors instead of a single color, to enable e.g. wild-card matches to accommodate for uncertainty.

[4] In precise terms, when the vertices that are part of at least one match induce a subgraph whose each connected component is a match. In this situation, one run of the vertex-localized sieve produces *all matches* to a query, with total effort that scales *linearly* in the number of edges m. Indeed, we first run the vertex-localized sieve, and then run, for example, a depth-first search on the vertices contained in at least one match to identify the connected components.

Al-Naffouri, Han, and Chung [25]. Here we look at implementation for small values of b and rely on independent repetitions of the sieve to decrease the probability of false negatives. Obtaining a high-performance implementation presents a minor engineering obstacle due to the fact that the instruction set of e.g. NVIDIA GPUs does not directly support multiplication in characteristic 2, whereas modern CPUs implement instruction set extensions with such support (cf. Gueron and Kounavis [16]). We rely on software techniques and use *bit-slicing* (cf. Biham [3] and Rudra, Dubey, Jutla, Kumar, Rao, and Rohatgi [34]) to multiply in parallel in units of 32 elements of $\mathrm{GF}(2^8)$ at a time using a simplified Mastrovito [26] multiplier implemented with Boolean word operations. We also experiment with other implementations for arithmetic in characteristic 2, but bit-slicing is clearly the fastest, yielding multiplication rates of more than 2.4 trillion $\mathrm{GF}(2^8)$-multiplications per second on an NVIDIA Tesla V100 SXM2 Accelerator, cf. Table 4 in the Appendix.

(iv) *Open-source implementation.* To encourage and ease further contributions, we release our implementation as open-source software under the MIT License.[5]

Earlier Work. Motif search on graphs is a hard generalization of jumbled pattern matching on strings (cf. [9, 14, 15, 18]) that was introduced by Lacroix, Fernandes, and Sagot [24] in a bioinformatics context. Multiple variants and extensions of the base variant studied in the present paper have been introduced and studied in a number of works, including Bruckner, Hüffner, Karp, Shamir, and Sharan [8], Dondi, Fertin, and Vialette [11], Pinter and Zehavi [32, 33, 37], Björklund, Kaski, Kowalik [6], Bonnet and Sikora [12], and Zehavi [38].

From the perspective of parameterized algorithms [10], Fellows, Fertin, Hermelin, and Vialette [13] established that the graph motif problem (parameterized by the motif size k) is fixed-parameter tractable using the color-coding technique of Alon, Yuster, and Zwick [1]. The scaling $f(k)$ as a function of the parameter k was improved in a sequence of works [2, 6, 17, 21, 31], including the randomized $O^*(2.54^k)$-time algorithm of Koutis [21], the randomized $O^*(2^k)$-time algorithm of Björklund, Kaski, and Kowalik [6], and the deterministic $O^*(5.22^k)$-time algorithm of Pinter, Scachnai, and Zehavi [31].[6]

Multilinear sieving and constrained multilinear sieving was developed in a sequence of works starting from pioneering work by Koutis [20], Williams [36], Koutis and Williams [22], and Koutis [21] on algebraic fingerprinting in group algebras (cf. Koutis and Williams [22]). The multivariate polynomial version of the sieve was developed by Björklund [4], Björklund, Husfeldt, Kaski, and Koivisto [5], and Björklund, Kaski, and Kowalik [6].

The generating polynomial P to capture connected sets of vertices in graphs can be traced back to Nederlof's [28] insight on branching walks for space-efficient algebraization of the Steiner tree problem. Guillemot and Sikora [17] transported this insight to the graph motif problem. The generating polynomial was further enhanced by Björklund, Kaski, and Kowalik [6], and Björklund, Kaski, Kowalik, and Lauri [7].

2 Scalar Recurrences for Sieving with Localization

This section details all the scalar recurrences in our vertex-localized algorithm, where by *scalar* we mean an element of the finite field $\mathbb{F}_{2^b}$ of size 2^b for a positive integer b. For an integer n, let us write $[n] = \{1, 2, \ldots, n\}$. This section only describes the algorithm;

[5] Available at: `https://github.com/pkaski/motif-localized`

[6] Here the asymptotic notation $O^*(\cdot)$ suppresses a multiplicative factor polynomial in the input size.

for reasons of space, we will include the mathematical derivation of the algorithm and its correctness analysis in an extended version of this work.

The input to the algorithm consists of a vertex-colored host graph H and a motif M. Here H is an undirected simple graph with vertex set $V(H)$ and edge set $E(H)$, the vertex-coloring is a function $c : V(H) \to C$ for a set of colors C, and the motif is a function $M : C \to \mathbb{Z}_{\geq 0}$ with $\sum_{q \in C} M(q) = k$ for a positive integer k.

For convenience in what follows, let us assume that the vertices of H are numbered $1, 2, \ldots, n$; that is, we assume that $V(H) = [n]$. Let us also introduce a set S_q of *shades* for each color $q \in C$, with $|S_q| = M(q)$ and $S_q \cap S_{q'} = \emptyset$ for all distinct $q, q' \in C$. For a vertex $i \in [n]$, let us write $\Gamma_H(i)$ for the set of vertices adjacent to i in H. Since H is simple, for each $j \in \Gamma_H(i)$ there is a unique edge $ij \in E(H)$ that joins i and j in H.

The algorithm now consists of the following five steps, where the key vectorizability property for implementation is highlighted in the main recurrence (d).

(a) Assign Random Values. First, draw an independent uniform random value $\mu_{i,d} \in \mathbb{F}_{2^b}$ for each $i \in [n]$ and $d \in S_{c(i)}$. Then, draw an independent uniform random value $\nu_{d,\ell} \in \mathbb{F}_{2^b}$ for each $d \in \cup_{q \in C} S_q$ and $\ell \in [k]$. Finally, draw an independent uniform random value $\alpha_{s,(i,j)} \in \mathbb{F}_{2^b}$ for each $s = 2, 3, \ldots, k$ and each orientation $(i, j) \in [n] \times [n]$ of an undirected edge $ij \in E(H)$ in H.

(b) Label Evaluation. For each $i \in [n]$ and $\ell \in [k]$, compute

$$\zeta_{i,\ell} = \sum_{d \in S_{c(i)}} \mu_{i,d} \nu_{d,\ell} \in \mathbb{F}_{2^b} . \tag{1}$$

We can parallelize this step over i and ℓ as appropriate.

(c) Initialize Label-Sum Vectors. For each subset $L \subseteq [k]$, compute the vector

$$\zeta^L = (\zeta_1^L, \zeta_2^L, \ldots, \zeta_n^L) \in \mathbb{F}_{2^b}^n \tag{2}$$

given for all $i \in [n]$ by

$$\zeta_i^L = \sum_{\ell \in L} \zeta_{i,\ell} \in \mathbb{F}_{2^b} . \tag{3}$$

We can parallelize this step over L and i as appropriate.

(d) The Main Recurrence. First, for $s = 1$ and each $i \in [n]$ and $L \subseteq [k]$, set

$$P_{i,1}(\zeta^L, \alpha) = \zeta_i^L . \tag{4}$$

Then, for each $s = 2, 3, \ldots, k$, $i \in [n]$, and $L \subseteq [k]$, compute

$$P_{i,s}(\zeta^L, \alpha) = \sum_{j \in \Gamma_H(i)} \alpha_{s,(i,j)} \sum_{\substack{s_1 + s_2 = s \\ s_1, s_2 \geq 1}} P_{i,s_1}(\zeta^L, \alpha) P_{j,s_2}(\zeta^L, \alpha) . \tag{5}$$

For each fixed value of s, we can parallelize this recurrence over i and L as appropriate. Furthermore, *the parallelization over L vectorizes*. That is, for each of the 2^k choices $L \subseteq [k]$, the index j ranges over precisely the same values in $\Gamma_H(i)$, and thus we can view the recurrence (5) as a recurrence *over vectors of length* 2^k, where the multiplication by $\alpha_{s,(i,j)}$ can be viewed as scalar-multiplication applied to a vector obtained as the sum of element-wise (Hadamard) products of vectors. This vectorizability is the gist of our GPU implementation described in the next section.

(e) Sum at Each Vertex. For each $i \in [n]$, take the sum over $L \subseteq [k]$ to obtain

$$Q_{i,k}(\mu, \nu, \alpha) = \sum_{L \subseteq [k]} P_{i,k}(\zeta^L, \alpha)\,. \tag{6}$$

We can parallelize this recurrence over i and over L as appropriate; parallelization over L can use routine parallel aggregation techniques.

This algorithm has the property that $Q_{i,k}(\mu, \nu, \alpha) = 0$ with probability 1 when there is no match to M in H that contains the vertex i, and $Q_i(\mu, \nu, \alpha) = 0$ with probability at most $(2k-1)/2^b$ when there is a match to M in H that contains the vertex i.

3 Engineering a Vector-Parallel Implementation

This section gives a high-level description of our implementation of the algorithm in §2. We intentionally avoid low-level details specific to a particular programming API such as CUDA for NVIDIA microarchitectures, with the understanding that the low-level details can be found in the accompanying source code. Our focus here is on principles that we believe generalize and support implementations on current and future vector-parallel architectures.

Workloads and Coalescence. It will be convenient to employ the following framework to describe at a high level how our implementation is structured.[7] Suppose we run a parallel workload that consists of work by W independent parallel threads, which have been arranged into a tensor of *volume* W with r *modes*, to obtain a tensor of *shape*[8]

$$W_r \times W_{r-1} \times \cdots \times W_1$$

for positive integers $W_1, W_2, \ldots, W_r$ with $W = W_r W_{r-1} \cdots W_1$.

Each thread $t = 0, 1, \ldots, W-1$ in this workload can now be identified with its r-tuple of coordinates $(t_r, t_{r-1}, \ldots, t_1)$ defined by $t = \sum_{j=1}^{r} t_j W_{j-1} W_{j-2} \cdots W_1$ and $t_j \in \{0, 1, \ldots, W_j - 1\}$ for all $j = 1, 2, \ldots, r$. In essence, the tuple $(t_r, t_{r-1}, \ldots, t_1)$ represents the integer t as a mixed-base r-digit integer in the number system defined by the sequence $(W_r, W_{r-1}, \ldots, W_1)$, where W_1 is the base of the least significant digit, W_2 is the base of the next least significant digit, and so on until W_r, which is the base of the most significant digit. To utilize vector-parallel hardware effectively, a key design principle is to ensure that the workload is *coalesced*, that is, any two threads t and t' that agree in all but possibly their c least significant coordinates are at all times executing the same instruction, and when this instruction is a memory access, the access is to units of memory at either *a single constant address* or *consecutive addresses* across the c least significant coordinates.[9]

Per-Thread Work Allocation and Memory Layout. When engineering an algorithm design for vector-parallel hardware with long latencies, among the most important considerations is to decide precisely what each thread in a parallel workload of W threads is going to do to

[7] A reader familiar e.g. with NVIDIA CUDA API should have no difficulty translating this framework to NVIDIA-specific terminology e.g. in terms of grids of thread blocks.

[8] We use the symbol "$\times$" to exclusively refer to Cartesian products and shapes of tensors, never for multiplication. For basic terminology on tensors, see Kolda and Bader [19].

[9] Precisely how large values c and $W_c W_{c-1} \cdots W_1$ one needs depends on the width of the hardware vectorization. For example, in case of NVIDIA microarchitectures, one usually wants $W_c W_{c-1} \cdots W_1$ to be a positive multiple of 32 to ensure coalescent execution of warps.

saturate the hardware and to hide latency (cf. Volkov [35]). Three key objectives underlying such a decision are to

- **(i)** expose sufficient parallelism in the design to enable a large W to saturate the hardware;
- **(ii)** ensure coalescent execution by a careful ordering of modes in the workload; and
- **(iii)** make sure each thread works with enough local data to make use of low-latency storage available to each thread and/or to select groups of threads.[10]

Interleaved with the question of per-thread work allocation is the question how to implement the memory layout of the algorithm across the memory hierarchy so that

- **(iv)** local storage with lowest latency is the most frequently accessed type of storage;
- **(v)** when accessing high-latency storage, as much data as possible should be accessed with a single access to saturate the pipeline; and
- **(vi)** accesses to memory are coalesced.

Design Choices for Vertex-Localized Sieving. Let us now turn to how we implement the recurrences in §2 as parallel workloads together with their memory layouts. For reasons of space, we focus only on the GPU-side workloads; the CPU-side workloads are those of Björklund, Kaski, Kowalik, and Lauri [7], with only minor modifications to support vertex-localization.

Top-Level Structure. We execute the random assignment step (a) and the label evaluation step (b) on the host CPU(s), with parallelization over i. The subsequent steps (c,d,e) each vector-parallelize over the 2^k evaluations indexed by $L \subseteq [k]$, and these steps will be offloaded for execution on the available GPU(s) using a sequence of parallel workloads.

Workloads on the GPU. Since label-sum initialization (c) and the main recurrence (d) both vector-parallelize over $L \subseteq [k]$, it is natural to design the workloads so that a divisor D of 2^k appears in the least significant mode of each workload to enable coalescent execution when k is large enough.[11] Furthermore, since we want simultaneous localization at each $i \in [n]$, it follows that n is a natural mode of the workload.[12] Indeed, although execution along this mode will not be coalesced, each of the D threads working with the same vertex will follow the same pattern of data-dependent memory accesses when traversing the adjacency list associated with the vertex i to obtain each vertex $j \in \Gamma_H(i)$ in (5). Finally, to ensure sufficient local data for low-latency computations (cf. (iii) in §3), we design the workload so that each thread that implements (5) works with S out of the 2^k scalars, where S divides 2^k. Thus, we will execute workloads of shape

$$n \times D\,,$$

where each thread will work with S scalars, so that SD divides 2^k.[13] When furthermore we have M devices (GPUs) available, where MSD divides 2^k, we execute workloads of shape

$$M \times n \times D\,.$$

[10] Here the registers locally available to each thread form the lowest-latency storage, whereas e.g. the shared memory available to each block of threads executing in a streaming multiprocessor in NVIDIA microarchitectures would form the next-lowest level of latency above the register file. Cf. Mei and Chu [27] for an empirical study of GPU memory architectures and their latency considerations.

[11] For example, on NVIDIA microarchitectures it is desirable to have $D \geq 32$.

[12] We furthermore assume that n has been rounded up to the closest power of 2 by inserting isolated vertices.

[13] For example, our bit-sliced implementation for arithmetic on $\mathrm{GF}(2^8)$ discussed in what follows assumes $S = 32$. With $D \geq 32$, we thus need $k \geq 10$ for coalescent execution.

with the mode of length M parallelized over the GPUs. That is, for each of the M GPUs we perform the following (asynchronously and in parallel on each GPU): First, we upload the evaluated labels (the output of (b)) from host memory to on-device memory on each GPU. Then, we iterate $\frac{2^k}{MSD}$ sequences of workloads, where each sequence consists of label-sum initialization (c) implemented with a single workload of shape $n \times D$, the main recurrence (d) implemented using a sequence of k workloads of shape $n \times D$, and the per-vertex sum (e) implemented as a standard batch-parallel sum that aggregates the nDS scalars output by (d) to n scalars. Finally, we download the n scalars to host memory. This results in Mn scalars downloaded from M GPUs in host memory, which we aggregate on the host to obtain n scalars, one scalar at each vertex, with parallelization over the vertices. The iteration over $\frac{2^k}{MSD}$ such sequences produces the final output (6) at each vertex $i \in [n]$.

Memory Layout on Each GPU. We now describe the memory layout used on each GPU. Since we execute workloads of $n \times D$ threads, and each thread works with S scalars, we want to access such S-scalar units of data with as-efficient-as-possible coalesced accesses (cf. §3 (v) and (vi)). Towards this end, suppose that S scalars occupy U words of memory and suppose the maximum amount of memory one thread can access (with a single load or store instruction) is A words, where A divides U. Then, to obtain coalesced memory accesses, we use a memory layout of shape

$$\frac{U}{A} \times n \times D \times A$$

and execute the loads/stores of S scalars per thread in groups of $\frac{U}{A}$ instructions that each load/store A words of data.[14] Adjacency lists of vertices and scalar associations with (oriented) edges are implemented as simple contiguous arrays of words. Due to the $n \times D$ workload to implement (5), each group of D threads working on a vertex $i \in [n]$ will access the same element j of an adjacency list or oriented-edge-associated scalar $\alpha_{s,(i,j)}$, implying coalescent execution for large enough D.

4 Experiments and Conclusion

This section reports on experiments with our algorithm implementation that extends the CPU implementation of Björklund, Kaski, Kowalik, and Lauri [7] with a vector-parallel GPU implementation of our vertex-localized sieve (cf. §2 and §3). Our implementation is prepared with CUDA C [29] using the OpenMP API [30] for host-side parallelization and to enable parallelization across multiple GPUs. The running times of CPU experiments are measured using OpenMP wall-clock time interface and the running time of GPU experiments are measured using CUDA event API. Memory usage is tracked using wrapper functions for the standard memory allocation interfaces. Memory bandwidth is tracked by computing the total amount of memory read/written in bytes by each recurrence and dividing by the measured running time. Arithmetic bandwidth is tracked by computing the number of scalar multiplications in each recurrence and dividing by the measured running time.

Our experiments use one CPU and four GPU configurations with a full hardware and software description provided in Appendix A. Here we give a short overview of the CPU node and the main GPU node in the experiments:

[14] In concrete terms, for NVIDIA microarchitectures and our bit-sliced arithmetic for GF(2^8) we choose $S = 32$, $A = 4$ and $U = 8$, measured in 32-bit words. That is, at CUDA level each thread reads/writes data in groups that consist of two coalesced `uint4`-accesses to global memory.

Table 1 Speedup obtained with a single GPU and eight GPUs compared with a CPU-only implementation as we increase the motif size k. We perform experiments on five independent d-regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \ldots, 20$. The CPU-only experiments are performed on the CPU compute node with the $64 \times \mathrm{GF}(2^8)$ bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node (single V100 device and eight V100 devices). All the running times are in seconds. The column "CPU" displays in each row the minimum time over the five graphs, whereas the columns "GPU V100" and "8 × GPU V100" displays in each row the maximum time over the five graphs. The column "Speedup (GPU)" displays the ratio of the columns "CPU" and "GPU V100", while the column "Speedup (Multi-GPU)" is the ratio of column "CPU" and "8 × GPU V100".

k	CPU	GPU V100	8 × GPU V100	Speedup (GPU)	Speedup (Multi-GPU)
10	0.0352 s	0.0432 s	0.0957 s	0.81	0.37
11	0.0828 s	0.0416 s	0.1180 s	1.99	0.70
12	0.1553 s	0.0696 s	0.0938 s	2.23	1.66
13	0.3808 s	0.0585 s	0.1046 s	6.51	3.64
14	0.7768 s	0.1062 s	0.1025 s	7.31	7.58
15	1.7244 s	0.1847 s	0.1111 s	9.33	15.52
16	3.9035 s	0.3968 s	0.1474 s	9.84	26.48
17	8.7340 s	0.8377 s	0.1906 s	10.43	45.82
18	19.3674 s	1.8950 s	0.3564 s	10.22	54.34
19	42.9873 s	4.1417 s	0.6480 s	10.38	66.34
20	94.2593 s	9.1468 s	1.2425 s	10.31	75.86

CPU compute node. An Apollo 6000 XL230a G9 blade server with two 2.6-GHz Intel Xeon E5-2690v3 CPUs (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache/CPU) and 128 GiB of main memory (8 × 16 GiB DDR4-2133 HP 752369-081).

V100 GPU compute node (NVIDIA DGX-1). An NVIDIA Tesla V100 SXM2 Accelerator device with one 1312-MHz NVIDIA GV100 GPU (Volta microarchitecture, 5120 cores, 80 SMs, 64 cores/SM) and 16384 MiB of on-device 4096-bit HBM2 with ECC enabled. The host is an NVIDIA DGX-1 with two 2.2-GHz Intel Xeon E5-2698v4 CPUs (Broadwell microarchitecture, 40 cores, 20 cores/CPU, hyper-threading enabled, 50 MiB L3 cache/CPU) and 512 GiB of main memory (16 × 32 GiB DDR4-2133 Samsung M393A4K40BB1-CRC). The host contains eight V100 devices.

Input Graphs. We use three synthetic graph topologies (regular, clique, and power-law) for our experiments, making use of the random graph generator from Björklund, Kaski, Kowalik, and Lauri [7]. The generator produces identical graph instances across all configurations to enable comparison between configurations. In addition, we use natural graph topologies obtained from the Koblenz Network Collection [23]. The following specific natural graphs from the Koblenz collection are considered in our experiments: Google [`web-Google`], Douban [`douban`], WordNet [`wordnet-words`], Stack Overflow [`stackexchange-stackoverflow`], Discogs [`discogs_affiliation`], MovieLens 10M [`movielens-10m_rating`], Hamsterster friendships [`petster-friendships-hamster`], Adolescent health [`moreno_health`], and Human protein (Stelzl) [`maayan-Stelzl`]; click on the text in brackets to follow the hyperlink. Each natural graph is preprocessed as in [7], i.e. (i) the vertices are randomly relabeled and (ii) to guarantee a unique match, a vertex is chosen uniformly at random and a monochromatic motif is placed on the first k vertices expanded by a depth-first search.

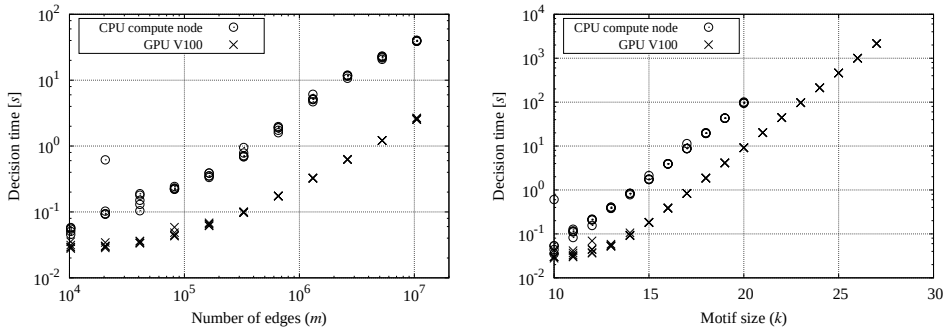

Figure 1 Scalability and speedup of the vertex-localized implementation. We compare the running time of CPU-only and single-GPU implementations as we increase the number of edges m (left) for five independent d-regular random graphs for each $n = 2^{10}, 2^{11}, \ldots, 2^{20}$, with $d = 20$ fixed and $k = 10$ fixed. We observe that our implementation scales linearly, as expected, with little variance between inputs except for small input sizes. For $n = 2^{20}$ with $k = 10$, our implementation on a single V100 GPU device is at least fourteen times faster than the CPU-only implementation on the CPU compute node. We compare the running time of the CPU-only and single-GPU implementations as we increase the motif size k (right) for five independent d-regular random graphs for each $n = 2^{20}$ fixed, $d = 20$ fixed and $k = 10, 11, \ldots, 30$. We observe that our implementation scales exponentially with respect to the motif size, as expected, with little variance between inputs. For $n = 2^{10}$ and $k = 20$, our implementation on a single V100 GPU device is at least ten times faster than the CPU-only implementation on the CPU compute node. The CPU-only experiments are executed on the CPU compute node with the $64 \times \mathrm{GF}(2^8)$ bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node, using a single V100 device.

Scalability and Speedup. Our first set of experiments studies the scalability of our implementation and compares our implementation with the CPU-only parallel implementation of Björklund, Kaski, Kowalik, and Lauri [7]. We report the running time as a function of, (i) number of edges m, and (ii) the motif size k. The results of the experiments are displayed in Figures 1 and 2. In Table 1, we observe speedups of several tenfolds for a sufficiently large k for the GPU and multi-GPU implementations over the CPU-only implementation. Table 5 in Appendix A reports the speedup of GPU and multi-GPU implementations with respect to the CPU-only implementation for increasing number of edges, while Table 6 considers the speedup of the multi-GPU over the GPU implementation for even larger k. We observe a substantial benefit both from (a) offloading from the host to a GPU device, and (b) offloading to multiple GPU devices compared with a single device.

Topology-Invariance. Our final set of experiments studies the effect of graph topology to scaling. Figure 3 displays the running times for regular, clique, power-law and natural topologies as a function of increasing m and increasing k. We observe that the running time is essentially invariant across topologies for increasing k. However, for increasing m, we observe differences in performance between topologies, with the worst performance occurring for power-law topologies with small exponent α. This is due to our design choice to use an $n \times D$ workload for the main recurrence (d), whereby the n groups of (length-D-vectorized) threads operating on different vertices have running times that are proportional to the degree of each vertex. When many groups of high-degree vertices get scheduled on the same streaming multiprocessor (SM) of the GPU, this produces an uneven distribution of work across SMs

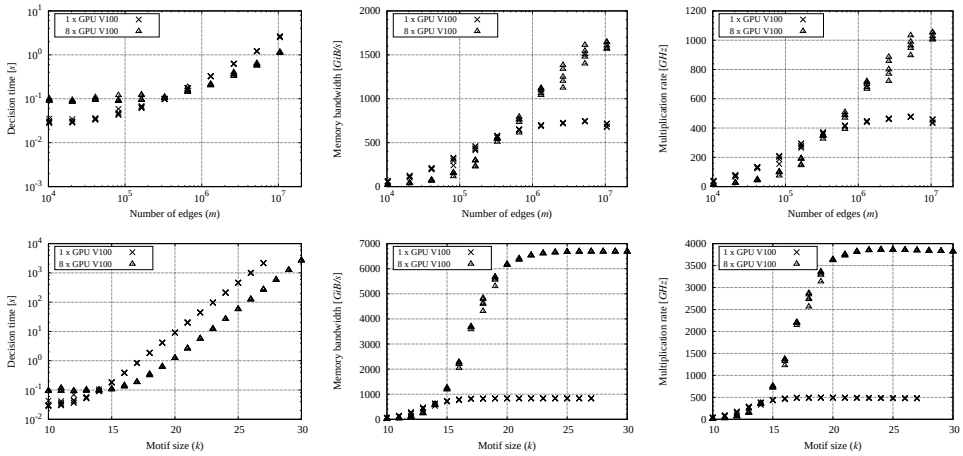

Figure 2 Scaling of our vertex-localized implementation as we increase the number of edges m (top row) and the motif size k (bottom row). For both rows, we display the runtime (left), memory bandwidth (middle), and arithmetic bandwidth (right). The experiments are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. **Top row** experiments are run for five independent d-regular random graphs for each $n = 2^{10}, 2^{11}, \ldots, 2^{10}$, $d = 20$ fixed and $k = 10$ fixed. For $n = 2^{20}$ with $k = 10$, the multi-GPU implementation executed on a configuration with eight V100 GPU devices is at least two times faster than the single-GPU implementation executed on a single V100 GPU device. We observe that the multi-GPU speedup becomes systematic only for large-enough m. For $n = 2^{20}$ and $k = 10$, with a single V100 device we obtain a memory bandwidth of at least 677 GiB/s and a simultaneous arithmetic bandwidth of more than 430 billion $\mathrm{GF}(2^8)$-multiplications per second. With eight V100 devices, we observe the results have more variance. For $n = 2^{20}$ and $k = 10$, we obtain at least 1567 GiB/s of memory bandwidth, and a simultaneous arithmetic bandwidth of at least 1003 billion $\mathrm{GF}(2^8)$-multiplications per second. **Bottom row** experiments are run for five independent d-regular random graphs for each $n = 2^{10}$, $d = 20$ fixed and $k = 10, 11, \ldots, 30$. For $n = 2^{10}$ and $k = 20$, our implementation on a single V100 GPU device is at least ten times faster than the CPU-only implementation on the CPU compute node. For $n = 2^{10}$ and $20 \leq k \leq 27$, the multi-GPU implementation executed on a configuration with eight V100 GPU devices is at least seven times faster than the single-GPU implementation executed on a single V100 GPU device. For $n = 2^{10}$ and $k = 27$, with a single V100 device we obtain a memory bandwidth of at least 835 GiB/s and a simultaneous arithmetic bandwidth of more than 480 billion $\mathrm{GF}(2^8)$-multiplications per second. With eight V100 devices, we obtain at least 6680 GiB/s of memory bandwidth, and a simultaneous arithmetic bandwidth of at least 3820 billion $\mathrm{GF}(2^8)$-multiplications per second. We observe that for large k these memory bandwidths noticeably *exceed the measured peak transfer bandwidth for on-device global memory* in Table 3, which we believe is caused by the relatively small value of n and in-SM-caching of the $P_{i,s_1}(\zeta^L, \alpha)$-values when executing the recurrence (5) across different $j \in \Gamma_H(i)$.

and delays the completion of the workload. Thus, our present implementation has topology-dependent performance, with the best performance obtained on graphs with a uniform distribution of vertex degrees. With nonuniform vertex degrees, a random permutation of the vertices (or, for example, a greedy bin-packing of the vertices by degree to the SMs) can be used to balance the load across SMs. Such load-balancing was not considered for the present implementation.

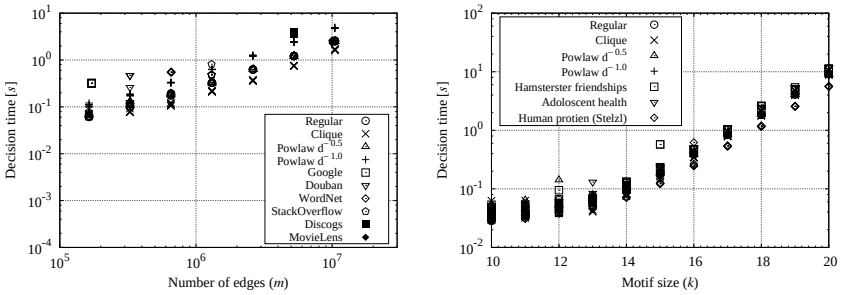

Figure 3 Topology-invariance. We display the runtime of our vertex-localized implementation with respect to number of edges (left) with different graph topologies: (a) five independent synthetic graphs for each value of n; and (b) five random relabelings for each natural graph. *Regular* graphs with $n = 2^{14}, 2^{15}, \ldots, 2^{20}$, $d = 20$ fixed; *cliques* with $n = 2^{13}, 2^{14}, \ldots, 2^{19}$, $d = 40$ fixed; *power-law* graphs with $n = 2^{14}, 2^{15}, \ldots, 2^{20}$, $D = 20$, $w = 100$ for both $\alpha = -0.5$ and $\alpha = -1.0$. All instances have $k = 10$. We display the runtime of our implementation with respect to the motif size (right) for different graph topologies: (a) five independent synthetic graphs for eack value of k; and (b) five random relabelings of natural graph for each value of k. *Regular* graphs with $n = 2^{10}$, $d = 20$; *cliques* with $n = 2^{10}$, $d = 40$; *power-law* graphs with $n = 2^{10}$, $D = 20$, $w = 100$ for both $\alpha = -0.5$ and $\alpha = -1.0$. The motif size varies from $k = 10, 11, \ldots, 20$. The experiments are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. All experiments use a single V100 device. For the motif-size scaling, the running times are essentially invariant, as expected. However, for scaling with respect to number of edges m, power-law graphs with $\alpha = -1.0$ consume approximately three times the running time of the clique graphs, and this ratio becomes more systematic with increasing m. In particular we observe that our present implementation is not completely topology-invariant; graphs with uniform degree distribution have relatively better performance than the graphs with non-uniform degree distribution (cf. §4).

More Data and Experiments in the Appendix. The Appendix contains a more extensive set of experiments.[15] For example, Figure 7 in Appendix A displays the running time of vertex-localization and decision-only variants. We observe that the overhead caused by vertex-localization is, as expected, negligible. In addition, the Appendix presents performance for our hardware configurations (Tables 2 and 3), arithmetic bandwidth measurements with different implementations of finite-field arithmetic (Table 4 and Figures 4, 5, and 6), and more detailed speedup tables (Tables 5 and 6).

Conclusion. This paper presented a vertex-localized variant of constrained multilinear sieving that enables algorithm engineering for vector-parallel microarchitectures such as single-GPU and multi-GPU configurations ranging from thousands to tens of thousands of cores, with tenfold to several tenfold speedups for large motif sizes compared with a carefully optimized parallel multi-CPU implementation [7]. The two key aspects of the present algorithm design that enable scalability are (i) the vectorization of the data-dependent memory accesses so that each traversal of an edge along an adjacency list results in memory accesses to long vectors of words at consecutive accesses when evaluating the main recurrence (cf. §2(d)), and (ii) the vectorization of the finite-field arithmetic in characteristic 2 through bit-slicing.

[15] *Caveat on false negatives.* The current set of experiments is still lacking an experiment that studies the empirical false negative rate compared with the theoretical per-vertex probability of at most $(2k-1)/2^b$ for a false negative. Such an experiment is warranted since our implementation uses $b = 8$ and thus independent repetitions of the sieve must be used when one seeks to control the false negative rate over the vertices.

References

1 Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

2 Nadja Betzler, Michael R. Fellows, Christian Komusiewicz, and Rolf Niedermeier. Parameterized algorithms and hardness results for some graph motif problems. In *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5029 of *Lecture Notes in Computer Science*, pages 31–43. Springer, 2008.

3 Eli Biham. A fast new DES implementation in software. In *Proceedings of the 4th International Workshop on Fast Software Encryption (FSE)*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 1997.

4 Andreas Björklund. Determinant sums for undirected hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014.

5 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Narrow sieves for parameterized paths and packings. *J. Comput. Syst. Sci.*, 87:119–139, 2017.

6 Andreas Björklund, Petteri Kaski, and Łukasz Kowalik. Constrained multilinear detection and generalized graph motifs. *Algorithmica*, 74(2):947–967, 2016.

7 Andreas Björklund, Petteri Kaski, Łukasz Kowalik, and Juho Lauri. Engineering motif search for large graphs. In *Proceedings of the 17th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 104–118. SIAM, 2015.

8 Sharon Bruckner, Falk Hüffner, Richard M. Karp, Ron Shamir, and Roded Sharan. Topology-free querying of protein interaction networks. *Journal of Computational Biology*, 17(3):237–252, 2010.

9 Ferdinando Cicalese, Travis Gagie, Emanuele Giaquinta, Eduardo Sany Laber, Zsuzsanna Lipták, Romeo Rizzi, and Alexandru I. Tomescu. Indexes for jumbled pattern matching in strings, trees and graphs. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 8214 of *Lecture Notes in Computer Science*, pages 56–63. Springer, 2013.

10 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms.* Springer, 2015.

11 Riccardo Dondi, Guillaume Fertin, and Stéphane Vialette. Maximum motif problem in vertex-colored graphs. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 5577 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 2009.

12 Édouard Bonnet and Florian Sikora. The graph motif problem parameterized by the structure of the input graph. *Discrete Applied Mathematics*, 231:78–94, 2017.

13 Michael R. Fellows, Guillaume Fertin, Danny Hermelin, and Stéphane Vialette. Upper and lower bounds for finding connected motifs in vertex-colored graphs. *J. Comput. Syst. Sci.*, 77(4):799–811, 2011. doi:10.1016/j.jcss.2010.07.003.

14 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. In *Proceedings of the 21st Annual European Symposium on Algorithms (ESA)*, volume 8125 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2013.

15 Emanuele Giaquinta and Szymon Grabowski. New algorithms for binary jumbled pattern matching. *Inf. Process. Lett.*, 113(14-16):538–542, 2013.

16 Shay Gueron and Michael E. Kounavis. *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode - Rev 2.02.* Intel Corporation, April 2014. [Link].

17 Sylvain Guillemot and Florian Sikora. Finding and counting vertex-colored subtrees. *Algorithmica*, 65(4):828–844, 2013.

18 Tomasz Kociumaka, Jakub Radoszewski, and Wojciech Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In *Proceedings of the 21st Annual*

European Symposium on Algorithms (ESA), volume 8125 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2013.

19 Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Rev.*, 51(3):455–500, 2009.

20 Ioannis Koutis. Faster algebraic algorithms for path and packing problems. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5125 of *Lecture Notes in Computer Science*, pages 575–586. Springer, 2008.

21 Ioannis Koutis. Constrained multilinear detection for faster functional motif discovery. *Inf. Process. Lett.*, 112(22):889–892, 2012. doi:10.1016/j.ipl.2012.08.008.

22 Ioannis Koutis and Ryan Williams. Algebraic fingerprints for faster algorithms. *Commun. ACM*, 59(1):98–105, 2016.

23 Jérôme Kunegis. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International World Wide Web Conference (WWW)*, pages 1343–1350, 2013. [Link].

24 Vincent Lacroix, Cristina G. Fernandes, and Marie-France Sagot. Motif search in graphs: Application to metabolic networks. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):360–368, 2006. doi:10.1109/TCBB.2006.55.

25 Sian-Jheng Lin, Tareq Y. Al-Naffouri, Yunghsiang S. Han, and Wei-Ho Chung. Novel polynomial basis with fast Fourier transform and its application to Reed-Solomon erasure codes. *IEEE Trans. Inform. Theory*, 62(11):6284–6299, 2016.

26 Edoardo Mastrovito. *VLSI Architectures for Computations in Galois Fields*. PhD thesis, Department of Electrical Engineering, Linköping University, 1991.

27 Xinxin Mei and Xiaowen Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans. Parallel Distrib. Syst.*, 28(1):72–86, 2017.

28 Jesper Nederlof. Fast polynomial-space algorithms using inclusion-exclusion. *Algorithmica*, 65(4):868–884, 2013.

29 NVIDIA Corporation. *CUDA C Programming Guide, Version 9.* [Link].

30 OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.5 – November 2015.* [Link].

31 Ron Y. Pinter, Hadas Shachnai, and Meirav Zehavi. Deterministic parameterized algorithms for the graph motif problem. *Discrete Applied Mathematics*, 213:162–178, 2016.

32 Ron Y. Pinter and Meirav Zehavi. Partial information network queries. In *Proceedings of the 24th International Workshop on Combinatorial Algorithms (IWOCA)*, volume 8288 of *Lecture Notes in Computer Science*, pages 362–375. Springer, 2013.

33 Ron Y. Pinter and Meirav Zehavi. Algorithms for topology-free and alignment network queries. *J. Discrete Algorithms*, 27:29–53, 2014.

34 Atri Rudra, Pradeep K. Dubey, Charanjit S. Jutla, Vijay Kumar, Josyula R. Rao, and Pankaj Rohatgi. Efficient Rijndael encryption implementation with composite field arithmetic. In *Proceedings of the 3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2001.

35 Vasily Volkov. *Understanding Latency Hiding on GPUs*. PhD thesis, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2016. Technical Report No. UCB/EECS-2016-143.

36 Ryan Williams. Finding paths of length k in $O^*(2^k)$ time. *Inf. Process. Lett.*, 109(6):315–318, 2009. doi:10.1016/j.ipl.2008.11.004.

37 Meirav Zehavi. Parameterized algorithms for module motif. In *Proceedings of the 38th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 8087 of *Lecture Notes in Computer Science*, pages 825–836. Springer, 2013.

38 Meirav Zehavi. Parameterized algorithms for the module motif problem. *Inf. Comput.*, 251:179–193, 2016.

A Additional Experimental Results

Additional Hardware Configurations. Our experiments use one CPU and four GPU configurations. The hardware configurations of CPU compute node and V100 GPU compute node are reported in Section 4. Each experiment documents the hardware configuration and the number of GPU devices used for the experiment.

K40 GPU compute node. An NVIDIA Tesla K40t Accelerator device with one 745-MHz NVIDIA GK110B GPU (Kepler microarchitecture, 2880 cores, 15 SMs, 192 cores/SM) and 12288 MiB of on-device GDDR5-3004 memory with ECC enabled. The host is a Bullx B715 DLC blade server with two 2.1-GHz Intel Xeon E5-2620v2 CPUs (Ivy Bridge microarchitecture, 12 cores, 6 cores/CPU, no hyper-threading, 15 MiB L3 cache) and 32 GiB of main memory (8 × 4 GiB DDR3-1600 Samsung M393B5273DH0-CMA). The host and the GPU are connected by a 16-lane PCI Express 3.0 bus. The host contains two K40t devices.

K80 GPU compute node. An NVIDIA Tesla K80 Accelerator device with two 875-MHz NVIDIA Tesla GK210 GPUs (Kepler microarchitecture, 2496 cores, 13 SMs, 192 cores/SM), 12288 MiB of on-device GDDR5-3004 memory with ECC enabled. The host is a Dell PowerEdge C4130 machine with two 2.4-GHz Intel Xeon E5-2620v3 CPUs (Haswell microarchitecture, 12 cores, 6 cores/CPU, no hyper-threading, 15 MiB L3 cache/CPU) and 128 GiB of main memory (16 × 8 GiB DDR4-2133 Hynix HMA42GR7AFR4N-TF). The host and the GPU are connected by a 16-lane PCI Express 3.0 bus. The host contains four K80 devices.

P100 GPU compute node. An NVIDIA Tesla P100 Accelerator device with one 1189-MHz NVIDIA GP100 GPU (Pascal microarchitecture, 3584 cores, 56 SMs, 64 cores/SM) and 16384 MiB of on-device 4096-bit HBM2 with ECC enabled. The host is a Dell PowerEdge C4130 with two 2.54-GHz Intel Xeon E5-2680v3 CPUs (Haswell microarchitecture, 24 cores, 12 cores/CPU, no hyper-threading, 30 MiB L3 cache/CPU) and 256 GiB of main memory (16 × 16 GiB DDR4-2133 Hynix HMA82GR7MFR8N-UH). The host and the device are connected by a 16-lane PCI Express 3.0 bus. The host contains four P100 devices.

Baseline Performance. Tables 2, 3, and 4 in Appendix A report the empirical baseline performance of the hardware. The host (CPU) memory bandwidth is measured by operating on a two gibibyte array of 64-bit words. Each experiment is executed five times and the average of five iterations is reported. Arithmetic bandwidth[16] is measured for different implementations of arithmetic in lines of S scalars per thread (cf. §3). The bit-sliced 32 × GF(2^8) line type has the best bandwidth, and this line type is used in our subsequent experiments. The on-device (GPU) and device–host (device-to-host and host-to-device) memory transfer rates are measured using the bandwidth-test tool distributed in the CUDA Examples package using a single device at a time.

[16] The number of scalar multiplications per second.

Table 2 Memory bandwidths of CPU compute node, P100 GPU compute node (host) and V100 compute node (host).

Benchmark	Single core	All cores
CPU compute node		
Read from linear addresses (consecutive 64-bit words)	9.13 GiB/s	46.93 GiB/s
Write to linear addresses (consecutive 64-bit words)	5.69 GiB/s	21.92 GiB/s
Read from random addresses (individual 64-bit words)	0.29 GiB/s	5.28 GiB/s
Read from random addresses (full cache lines)	1.46 GiB/s	19.94 GiB/s
P100 GPU compute node (host memory)		
Read from linear addresses (consecutive 64-bit words)	10.34 GiB/s	41.07 GiB/s
Write to linear addresses (consecutive 64-bit words)	8.43 GiB/s	22.91 GiB/s
Read from random addresses (individual 64-bit words)	0.52 GiB/s	5.53 GiB/s
Read from random addresses (full cache lines)	1.55 GiB/s	19.80 GiB/s
V100 GPU compute node (host memory)		
Read from linear addresses (consecutive 64-bit words)	9.58 GiB/s	36.39 GiB/s
Write to linear addresses (consecutive 64-bit words)	7.78 GiB/s	19.27 GiB/s
Read from random addresses (individual 64-bit words)	0.46 GiB/s	5.20 GiB/s
Read from random addresses (full cache lines)	1.45 GiB/s	20.14 GiB/s

Table 3 Memory bandwidth of the P100 and V100 GPU compute nodes, using one of the four P100 devices and one of the eight V100 devices, respectively.

Node	Host to device	Device to host	Device to device
P100 GPU compute node	12.16 GiB/s	12.87 GiB/s	498.15 GiB/s
V100 GPU compute node	10.79 GiB/s	12.17 GiB/s	720.77 GiB/s

Table 4 Arithmetic bandwidth of the P100 and V100 GPU compute nodes with different line implementations, using one of the P100 devices and one of the eight V100 devices, respectively.

Line type	Line multiplication	Scalar multiplication
P100 GPU		
$1 \times \mathrm{GF}(2^8)$ bit-packed line	82.04 GHz	82.68 GHz
$4 \times \mathrm{GF}(2^8)$ bit-packed line	328.20 GHz	331.09 GHz
$16 \times \mathrm{GF}(2^8)$ bit-packed line	348.30 GHz	348.30 GHz
$32 \times \mathrm{GF}(2^8)$ bit-sliced line	1462.62 GHz	1500.70 GHz
$1 \times \mathrm{GF}(2^8)$ lookup table	65.23 GHz	85.58 GHz
$4 \times \mathrm{GF}(2^8)$ lookup table	64.59 GHz	86.64 GHz
$16 \times \mathrm{GF}(2^8)$ lookup table	64.56 GHz	84.54 GHz
$32 \times \mathrm{GF}(2^8)$ lookup table	64.80 GHz	84.52 GHz
V100 GPU		
$1 \times \mathrm{GF}(2^8)$ bit-packed line	159.58 GHz	159.72 GHz
$4 \times \mathrm{GF}(2^8)$ bit-packed line	638.25 GHz	638.88 GHz
$16 \times \mathrm{GF}(2^8)$ bit-packed line	652.86 GHz	723.69 GHz
$32 \times \mathrm{GF}(2^8)$ bit-packed line	661.59 GHz	719.31 GHz
$32 \times \mathrm{GF}(2^8)$ bit-sliced line	2486.54 GHz	2441.10 GHz
$1 \times \mathrm{GF}(2^8)$ lookup table	410.52 GHz	496.94 GHz
$4 \times \mathrm{GF}(2^8)$ lookup table	408.91 GHz	510.29 GHz
$16 \times \mathrm{GF}(2^8)$ lookup table	409.91 GHz	531.27 GHz
$32 \times \mathrm{GF}(2^8)$ lookup table	409.93 GHz	528.28 GHz

Table 5 Speedup obtained with a single GPU and eight GPUs compared with a CPU-only implementation as we increase the number of edges m. We perform experiments on five independent random d-regular graphs for each $n = 2^{10}, 2^{11}, \ldots, 2^{20}$, with $d = 20$ fixed and $k = 10$ fixed. The CPU-only experiments are performed on the CPU compute node with the 64 × GF(2^8) bit-packed line type configured for the Björklund–Kaski–Kowalik–Lauri [7] implementation without vertex-localization. The GPU experiments with our implementation are configured with the 32 × GF(2^8) bit-sliced line type and executed on the V100 GPU compute node (single V100 device and eight V100 devices). All the running times are in seconds. The column "CPU" displays in each row the minimum time over the five graphs, whereas the columns "GPU V100" and "8 × GPU V100" displays in each row the maximum time over the five graphs. The column "Speedup (GPU)" displays the ratio of the columns "CPU" and "GPU V100", while the column "Speedup (Multi-GPU)" is the ratio of column "CPU" and "8 × GPU V100".

n	CPU	GPU V100	8 × GPU V100	Speedup (GPU)	Speedup (Multi-GPU)
2^{10}	0.0456 s	0.0353 s	0.1009 s	1.29	0.45
2^{11}	0.0923 s	0.0344 s	0.0924 s	2.68	1.00
2^{12}	0.1042 s	0.0363 s	0.1081 s	2.87	0.96
2^{13}	0.2199 s	0.0586 s	0.1204 s	3.75	1.83
2^{14}	0.3339 s	0.0685 s	0.1242 s	4.88	2.69
2^{15}	0.6859 s	0.1014 s	0.1109 s	6.76	6.18
2^{16}	1.5949 s	0.1752 s	0.1842 s	9.10	8.66
2^{17}	4.7021 s	0.3292 s	0.2171 s	14.28	21.66
2^{18}	10.6986 s	0.6302 s	0.4022 s	16.98	26.60
2^{19}	20.7284 s	1.2187 s	0.6465 s	17.01	32.06
2^{20}	38.8648 s	2.6742 s	1.1552 s	14.53	33.64

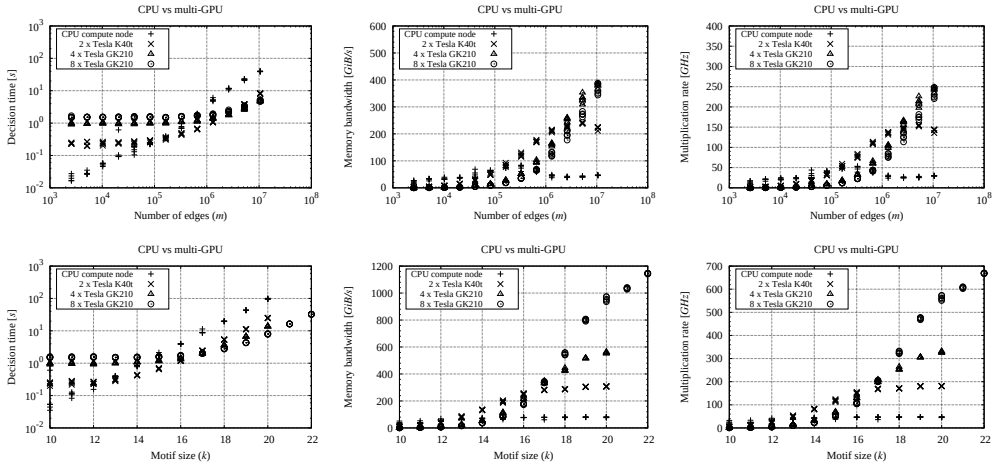

Figure 4 Performance of the vertex-localized implementation in older GPU microarchitectures as we increase, (i) the number of edges m, and (ii) the motif size k. In the top row, we display the running time (left), memory bandwidth (center) and arithmetic bandwidth (right) for five independent d-regular random graphs with $n = 2^{10}, 2^{11}, \ldots, 2^{10}$, $d = 20$ fixed and $k = 10$ fixed. In bottom row, we display the running time (left), memory bandwidth (center) and arithmetic bandwidth (right) for five independent d-regular random graphs with $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \ldots, 22$. Each configuration is reserved exclusively for the experiments at hand. The GPU experiments with our implementation are configured with the 32 × GF(2^8) bit-sliced line type. The CPU-only experiments use the Björklund–Kaski–Kowalik–Lauri [7] implementation with the 64 × GF(2^8) bit-packed line type.

Table 6 Speedup obtained with eight GPUs compared with a single GPU as we increase the motif size k. We perform experiments on five independent graph inputs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 21, 22, \ldots, 30$. The GPU experiments with our implementation are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the V100 GPU compute node. All the running times are in seconds. The column "GPU V100" displays in each row the minimum time over the five graphs when using a single V100 device, whereas the column "8 × GPU V100" displays in each row the maximum time over the five graphs when using four V100 devices. The column "Speedup" displays the ratio of the columns "GPU V100" and "8 × GPU V100".

k	GPU V100	8 × GPU V100	Speedup
21	20.1962 s	2.6556 s	7.61
22	44.4414 s	5.6852 s	7.82
23	97.2945 s	12.3074 s	7.91
24	212.3904 s	26.6797 s	7.96
25	461.7525 s	57.8421 s	7.98
26	1000.1718 s	125.0891 s	8.00
27	2160.4430 s	270.0623 s	8.00
28	-	581.5915 s	-
29	-	1249.4652 s	-
30	-	2676.9140 s	-

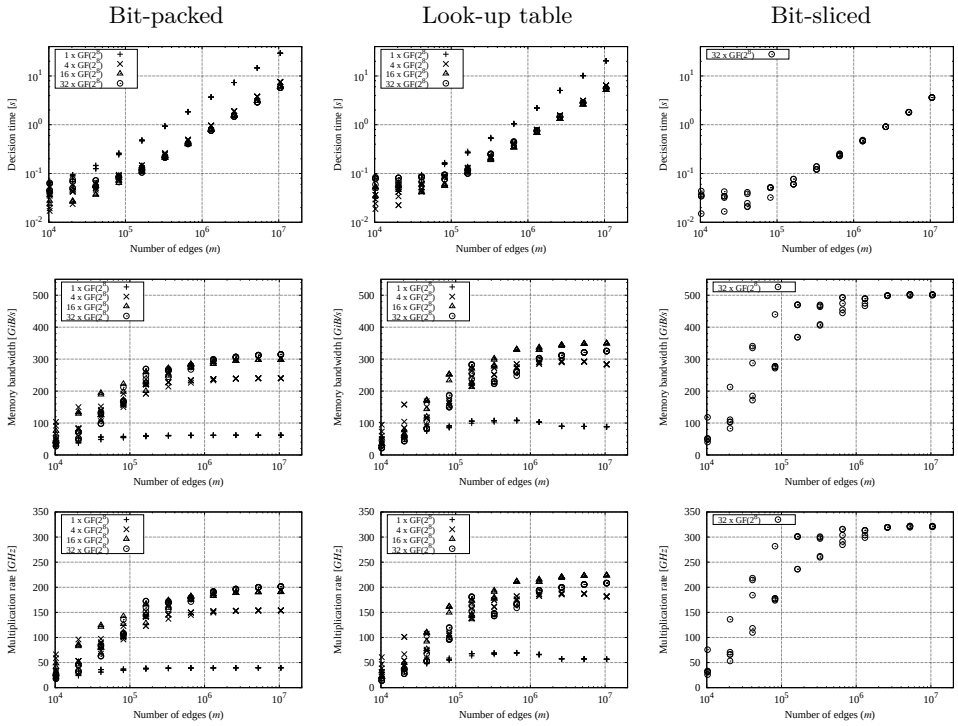

Figure 5 Performance comparison of different scalar line types with increasing number of edges m. We display the runtime (top row), memory bandwidth (middle row) and arithmetic bandwidth (bottom row) of five independent d-regular random graphs for each $n = 2^8, 2^9, \ldots, 2^{20}$, $d = 20$ fixed and $k = 10$ fixed. The experiments are configured with each line type and executed on the P100 GPU compute node. All experiments use a single P100 device. The bit-sliced $32 \times \mathrm{GF}(2^8)$ line type has the best performance.

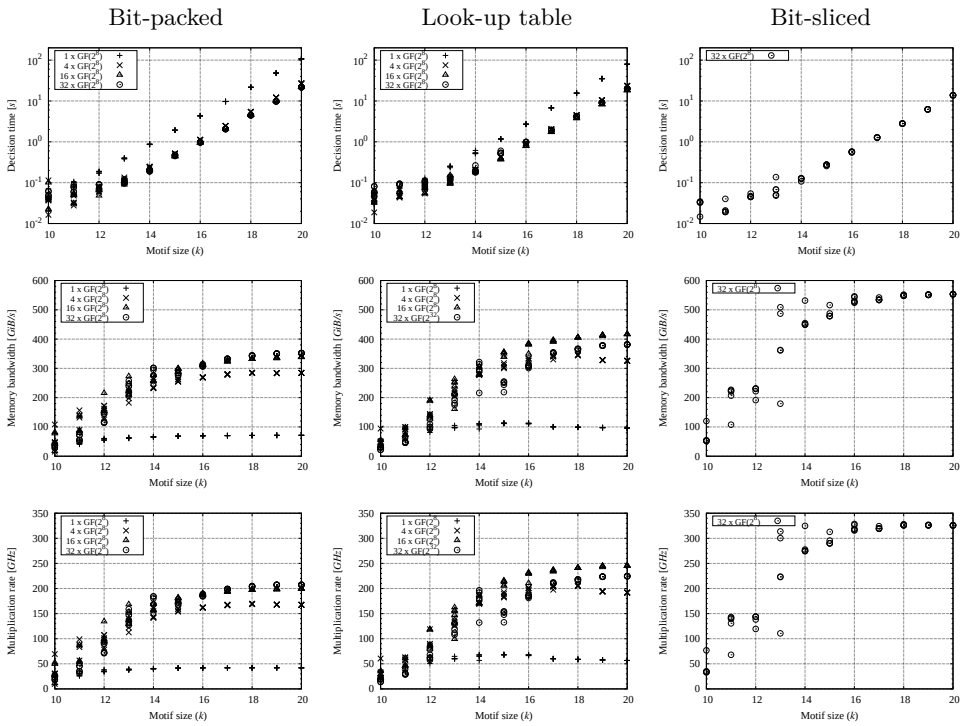

Figure 6 Performance comparison of different scalar line types with increasing motif size k. We display the runtime (top row), memory bandwidth (middle row) and arithmetic bandwidth (bottom row) for five independent d-regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \ldots, 20$. The experiments are configured with each line type and executed on the P100 GPU compute node. All experiments use a single P100 device. The bit-sliced $32 \times \mathrm{GF}(2^8)$ line type has the best performance.

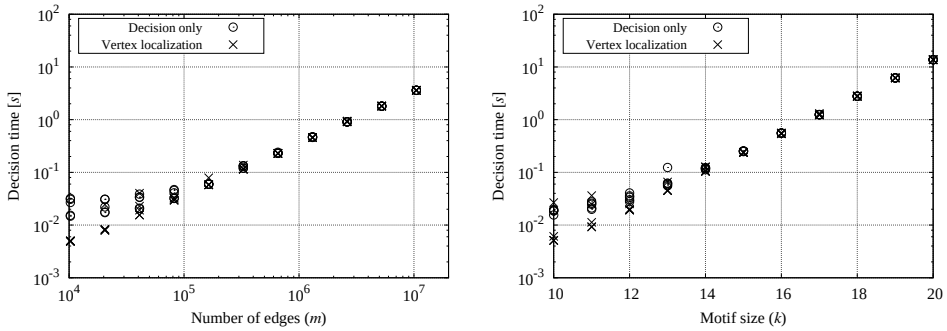

Figure 7 Overhead of vertex-localization. Here we compare our vertex-localized implementation against a separately prepared GPU implementation that uses the original Björklund–Kaski–Kowalik–Lauri design [7] without vertex-localization. We show scaling as a function of the number of edges (left) and the motif size (right). The left plot is the running time of five independent d-regular random graphs for each configuration of $n = 2^{10}, 2^{11}, \ldots, 2^{20}$, $d = 20$ fixed and $k = 10$ fixed. The right plot is the running time of five independent d-regular random graphs for each $n = 2^{10}$ fixed, $d = 20$ fixed and $k = 10, 11, \ldots, 20$. The experiments are configured with the $32 \times \mathrm{GF}(2^8)$ bit-sliced line type and executed on the P100 GPU compute node. All experiments use a single P100 device. We observe that the overhead of vertex-localization is negligible, as expected.