UC Riverside

UC Riverside Previously Published Works

Title
High performance FPGA and GPU complex pattern matching over spatio-temporal streams

Permalink
https://escholarship.org/uc/item/98d401z2

Authors
Moussalli, R
Absalyamov, I
Vieira, MR
et al.

Publication Date
2014-08-26

DOI
10.1007/s10707-014-0217-3

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Geoinformatica
DOI 10.1007/s10707-014-0217-3

High performance FPGA and GPU complex pattern matching over spatio-temporal streams

Roger Moussalli · Ildar Absalyamov · Marcos R. Vieira · Walid Najjar · Vassilis J. Tsotras

Received: 30 January 2014 / Revised: 15 June 2014 / Accepted: 11 August 2014
© Springer Science+Business Media New York 2014

Abstract The wide and increasing availability of collected data in the form of trajectories has led to research advances in behavioral aspects of the monitored subjects (e.g., wild animals, people, and vehicles). Using trajectory data harvested by devices, such as GPS, RFID and mobile devices, complex pattern queries can be posed to select trajectories based on specific events of interest. In this paper, we present a study on FPGA- and GPU-based architectures processing complex patterns on streams of spatio-temporal data. Complex patterns are described as regular expressions over a spatial alphabet that can be implicitly or explicitly anchored to the time domain. More importantly, variables can be used to substantially enhance the flexibility and expressive power of pattern queries. Here we explore the challenges in handling several constructs of the assumed pattern query language, with a study on the trade-offs between expressiveness, scalability and matching accuracy. We show an extensive performance evaluation where FPGA and GPU setups outperform the current state-of-the-art (single-threaded) CPU-based approaches, by over three orders of magnitude for FPGAs (for expressive queries) and up to two orders of magnitude for certain datasets on GPUs (and in some cases slowdown). Unlike software-based approaches,

R. Moussalli (✉)
IBM T.J. Watson Research Center, 1101 Kitchawan Rd., Yorktown Heights, NY, USA
e-mail: rmoussal@us.ibm.com

I. Absalyamov · W. Najjar · V. J. Tsotras
Department of Computer Science and Engineering, University of California, Riverside, 900 University Ave., Riverside, CA, USA
e-mail: iabsa001@cs.ucr.edu

W. Najjar
e-mail: najjar@cs.ucr.edu

V. J. Tsotras
e-mail: tsotras@cs.ucr.edu

M. R. Vieira
IBM Research - Brazil, Av. Pasteur 138, 146, Rio de Janeiro, RJ, Brazil
e-mail: mvieira@br.ibm.com

Published online: 26 August 2014

the performance of the proposed FPGA and GPU solutions is only minimally affected by the increased pattern complexity.

Keywords Spatio-temporal · Spatial · Temporal · Database · FPGA · GPU · Acceleration · Pattern · Matching

1 Introduction

Due to their relative ease of use, general purpose processors are commonly favored at the heart of many computational platforms. These processors are deployed in environments with varying requirements, ranging from personal electronics to game consoles, and up to server-grade machines. General purpose CPUs follow the Von-Neumann model, which executes instructions sequentially. Nevertheless, in this model performance does not always linearly scale in multi-processor environments, mostly due to the challenges of data sharing across cores. As it is non-trivial for these CPUs to satisfy the increasing time-critical demands of several applications, they are often coupled with application- or domain-specific parallel accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), which strive given a certain class of instructions and memory access patterns.

FPGAs consist of a fully configurable hardware platform, providing the flexibility of software (e.g., programmability) and the performance benefits of hardware (e.g., parallelism). The performance advantages of such platforms arise from their ability to execute thousands of parallel computations, relieving the application at hand from the sequential limitations of software execution on Von-Neumann based platforms. The processor "instructions" are now the logic functions processing the input data. Depending on the application, one big advantage of FPGAs is the ability to process streaming data at wire speed, thus resulting in a minimal memory footprint. The aforementioned advantages are shared with Application Specific Integrated Circuits (ASIC). FPGAs, however, can be reconfigured and are more adaptable to changes in applications and specifications, and hence exhibit a faster time to market. This comes at a slight cost in performance and in area, where one functional circuit would run faster on a tailored ASIC and require fewer gates.

For a long time GPUs were considered to be hardware devices specifically for graphics-related computations. However recent advances in general-purpose GPU computing (GPGPU) enabled application developers to use excessive computational power of GPUs for ad-hoc processing. Although programming GPUs remains a software task, the architecture of GPUs, being completely different from general-purpose central processors, allows a programmer to achieve greater performance by leveraging very large scale parallelism. This architecture, highly tailored towards parallelism, dictates a specific programming approach and imposes some limitations, which should be addressed by the application running on the GPU. Typically the performance achieved by using parallel GPU architecture is inferior to the performance achieved on ASICs or FPGAs for a similar task. However, unlike an FPGA-based solution, the program running on a GPU could be changed in an *online fashion*, thus providing excellent adaptivity.

As traditional platforms are increasingly hitting limitations when processing large volumes of streaming data, researchers have been investigating FPGAs for database applications. Recent work has focused on the adoption of FPGAs for data stream processing in different scenarios. In [23] a stream filtering approach is presented for XML documents. [37] investigated the speedup of the frequent item problem using FPGAs. In [40], the

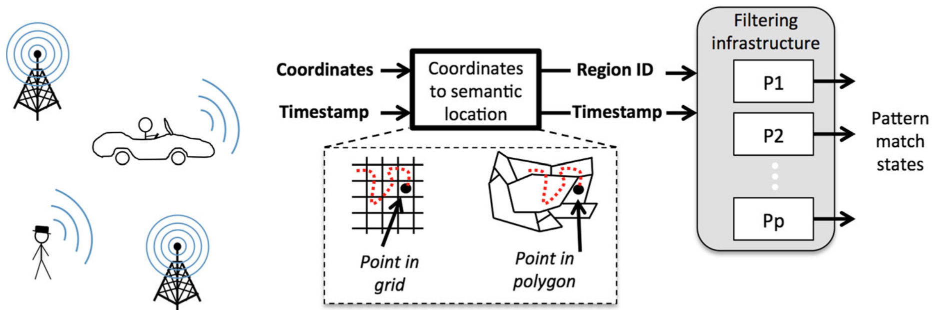


Fig. 1 Generic overview of various steps performed in spatio-temporal querying setups

FPGA is employed for complex event detection using regular expressions. [31] proposed a predicate-based filtering on FPGAs where user profiles are expressed as a conjunctive set of boolean filters. [21] describes an FPGA-based stream-mode decompression engine targeting Golomb-Rice encoded inverted indexes.

Similar research has been conducted on using GPUs for accelerating typical database operations. Some of them focus on improving performance of relational operators, e.g., selections and projections [3] and joins [12]. Recently, GPU architectures have been used to implement tree index operations [4, 14], as well as filtering for XML Path/Twig queries [1, 20].

In this paper, we describe both FPGA- and GPU-based solutions for querying trajectory data using patterns. Pattern matching allows users to query spatio-temporal databases in a very powerful and intuitive way [8, 10, 26, 32, 38]. Figure 1 describes the setup. Streams of trajectory data are harvested from devices, such as GPS and cellular devices. Coordinates are then translated into semantic regions that partition the spatial domain; these regions can be grid regions representing areas of interests (e.g., neighborhoods, school districts, cities). Our work is based on the FlexTrack framework [38, 39], which allows users to query trajectory databases using flexible patterns. A flexible pattern query is specified as a combination of sequential spatio-temporal predicates, allowing the end user to search for specific parts of interests in trajectory databases. For example, the pattern query "*Find all taxi cabs (trajectories) that first were in downtown Munich in the morning, later passed by the Olympiapark around noon, and then were closest to the Munich airport*" provides a combination of temporal, range and Nearest-Neighbor (NN) predicates that have to be satisfied in the specific order. Essentially, flexible patterns cover that part of the query spectrum between the single spatio-temporal predicate queries, such as the range predicate covering certain time instances of the trajectory life (e.g., "*Find all trajectories that passed by the Deutsches Museum area at 11pm*"), and similarity/clustering based queries, such as extracting similar movement patterns from a trajectories that cover the entire life span of the trajectory (e.g., "*Find all trajectories that are similar to a given query trajectory according to some similarity measure*").

The *FlexTrack* framework provides support for "variable" spatial predicates, which substantially enhances the flexibility and expressive power of the pattern queries. An example of a variable-enhanced query is "*Find all trajectories that started in a region @x, then visited the downtown Munich, then at some later point returned to region @x*".

This paper serves as a proof-of-concept on the performance benefits of evaluating flexible pattern queries using FPGAs and GPUs. A preliminary version appeared in [24], where

only the FPGA-based setup was considered. Our focus is on the challenges presented when supporting hundreds (up to thousands) of variable-enhanced flexible patterns in a streaming (fully-pipelined) fashion. Using both hardware platforms all pattern query predicates are evaluated in parallel over sequential streams of trajectories, hence resulting in over three orders of magnitude speedup over CPU-based approaches for FPGAs, and up to 2 times for GPUs. This performance property also holds even when compared to CPU-based setups where the pre-processing of trajectories is performed beforehand using specialized indexes. To the best of our knowledge, this work is the first detailing FPGA **and** GPU support for flexible pattern queries.

The remainder of this paper is organized as follows: related work is described in Section 2; the *FlexTrack* query language is detailed in Section 3; the proposed FPGA-based and GPU-based querying architecture is detailed in Section 4 and Section 5, respectively; the experimental evaluation is provided in Section 6; and Section 7 concludes this paper.

2 Related work

Single predicate queries (e.g., Range and *NN* queries) for trajectory data have been widely studied (e.g., [2, 28, 36]). In order to make the query evaluation process more efficient [11], trajectories are first approximated using Minimum Bounding Regions (MBR) and then indexed using hierarchical spatio-temporal indexing structures, like the MVR-tree [35]. However, these solutions are only efficient to evaluate single predicate queries. For moving object data, patterns have been examined in the context of query language and modeling issues [8, 19, 32], as well as query evaluation algorithms [10, 25].

The *FlexTrack* system [38, 39], on which our work is based, provides a general and powerful query framework. In *FlexTrack*, queries can contain both fixed and *variable* regions, as well as regular expression structures (e.g., repetitions, negations, optional structures) and explicit ordering of the predicates along the temporal dimension. This system uses a hierarchical region alphabet, where the user has the ability to define queries with finer alphabet granularity (*zoom in*) for the portions of greater interest, and higher granularity (*zoom out*) elsewhere. In order to efficiently evaluate flexible pattern queries, *FlexTrack* employs two lightweight index structures in the form of ordered lists in addition to the raw trajectory data. Given these index structures four different algorithms for evaluating flexible pattern queries are available, which are detailed in the next section.

The use of hardware platforms for pattern matching has been explored by many studies [17, 18, 34, 40]. Most of these works focus on deep packet inspection and security as applications of interest. Speed-ups over CPU-based approaches of up to two orders of magnitude have been reported using FPGAs: every cycle a new data element can be processed from the stream. The works in [1, 20, 22, 23] present a novel dynamic programming, push down automata approach, using FPGAs and GPUs, for matching XML Path and Twig patterns in XML documents. Using the massively parallel solution running on parallel platforms, up to three orders of magnitude speedup was achieved versus state-of-the-art CPU-based approaches.

In [34] an NFA implementation of regular expressions on FPGAs is described. A pattern matching approach, built on GPU-based NFA regular expression engine is reported in [5]. Generating hardware code from Perl Compatible Regular Expressions (PCRE) is proposed in [18] . The work in [17] focuses on DFA implementations of regular expressions, while merging commonalities among multiple DFAs. The use of regular expressions for the representation of spatio-temporal queries is proposed in [40] where an FPGA implementation is

detailed, allowing the sharing of query evaluation engines among several trajectories, with a minor impact on performance. While simpler query matching engines are assumed (due to less expressive queries), the mechanism of sharing engines for the purpose of frequently updated trajectories can also be applied to the approach described in this paper (where batch analytics are the main focus). The use of GPUs for the fast computation of proximity area views over streams of spatio-temporal data is investigated in [6]. Our work mainly differs from all the above works from the perspective of the query language, described in Section 3. Specifically, we describe an investigation of the FPGA- and GPU-based support of variable-enhanced patterns.

3 The *FlexTrack* system

We now provide a brief description of the pattern query language syntax, as well as the key elements in the *FlexTrack* framework (see [38] for more details).

3.1 Flexible pattern query language

The *FlexTrack* uses a set of non-overlapping regions Σ_l that are derived from partitioning the spatial domain into l regions. Such regions correspond to areas of interest (e.g. *school districts*, *airports*) and form the alphabet language $\Sigma = \bigcup_l \Sigma_l = \{A, B, C, ...\}$. The *FlexTrack* query language defines a spatio-temporal predicate $\mathcal{P}$ by a triplet $\langle op, \mathcal{R}[, t]\rangle$, where $\mathcal{R}$ corresponds to a predefined spatial region in Σ or a *variable* in Γ ($\mathcal{R} \in \{\Sigma \cup \Gamma\}$), *op* describes the topological relationship (e.g. *meet*, *overlap*, *inside*) that the trajectory and the spatial region $\mathcal{R}$ must satisfy over the (optional) time $t := (t_{from} : t_{to}) \mid t_s \mid t_r$ (t_{from}:t_{to}: time interval; t_s: snapshot time; t_r: relative time to a previous match time predicate). A predefined spatial region is explicitly specified by the user in the query predicate (e.g. "the downtown area of Munich"). In contrast, a *variable* denotes an arbitrary region using the symbols in $\Gamma = \{@x, @y, @z, ...\}$. Conceptually, *variables* work as placeholders for explicit spatial regions and can be bound to a specific region during the query evaluation.

The *FlexTrack* language defines a pattern query $\mathcal{Q} = (\mathcal{S}\ [\cup\ \mathcal{D}])$ as a combination of a sequential pattern $\mathcal{S}$ and an optional set of constraints $\mathcal{D}$. A trajectory matches $\mathcal{Q}$ if it satisfies both $\mathcal{S}$ and $\mathcal{D}$ parts. The $\mathcal{D}$ part of $\mathcal{Q}$ allows us to describe general constraints. For instance, constraints can be distance-based constraints among the *variables* in $\mathcal{S}$ and the predefined regions in Σ. And $\mathcal{S} := \mathcal{S}.\mathcal{S} \mid \mathcal{P} \mid !\mathcal{P} \mid \mathcal{P}^{\#} \mid ?^{+} \mid ?^{*}$ corresponds to a sequence of spatio-temporal predicates, while $\mathcal{D}$ represents a collection of constraints that may contain regions defined in $\mathcal{S}$ ("!" and "#" define negation and optional operators, respectively). The wild-card ? is also considered a variable, however it refers to any region in Σ, and not necessarily the same region if it occurs multiple times within a pattern $\mathcal{S}$.

The use of the same set of *variables* in describing both the topological predicates and the numerical conditions provides a very powerful language to query trajectories. To describe a query in *FlexTrack*, the user can use fixed regions for the parts of the trajectory where the behavior should satisfy known (strict) requirements, and *variables* for those sections where the exact behavior is not known but can be described by *variables* and the constraints between them.

In addition to the query language defined previously, we introduce the variable region set constraint defined in $\mathcal{D}$. A region set constraint (e.g., $\{@x : A, D, E\}$) is optional per variable, and can be only applied to variable predicates, having the purpose of limiting the region values that a given variable can take in Σ. Consider the following query pattern

and region set over $@x$, $\mathcal{Q} = (\mathcal{S} = \{A.B.@x.C.?^{+}.@x\},\ \mathcal{D} = \{@x : A, D, E\})$. Here, $@x$ is constrained by the regions $\{A, D, E\}$. In practice, a variable can be limited to the neighboring regions of the fixed query predicates. Other constraints can be set by the user, hence, limiting the number of matches of interest. From a performance perspective, the use of variable region set constraints greatly simplifies hardware support for variable predicates separated by wildcards $?^{+}$ or ?*, as detailed in Section 4.

3.2 Flexible pattern query evaluation

The *FlexTrack* system employs two lightweight index structures in the form of ordered lists that are stored in addition to the raw trajectory data. There is one *region-list* (*R-list*) per region in Σ, and one *trajectory-list* (*T-list*) per trajectory in the database. The *R-list* $\mathcal{L}_{\mathcal{I}}$ of a given region $\mathcal{I} \in \Sigma$ acts as an inverted index that contains all trajectories that passed by region $\mathcal{I}$. Each entry in $\mathcal{L}_{\mathcal{I}}$ contains a trajectory identifier T_{id}, the time interval (*ts-entry*:*ts-exit*] during which the trajectory was inside $\mathcal{I}$ (*ts-entry* included, *ts-exit* excluded), and a pointer to the *T-list* of T_{id}. Entries in a *R-list* are ordered first by T_{id}, and then by *ts-entry*.

The *T-list* is used to fast prune trajectories that do not satisfy pattern $\mathcal{S}$. For each trajectory T_{id} in the database, the *T-list* is its approximation represented by the regions it visited in the partitioning space Σ. Each entry in the *T-list* of T_{id} contains the region and the time interval (*ts-entry*:*ts-exit*] during which this region was visited by T_{id}, ordered by *ts-entry*. In addition, entries in *T-list* maintain pointers to the *ts-entry* part in the original trajectory data. With the above described index structures, there are four different strategies for evaluating flexible pattern queries:

1. *Index Join Pattern* (*IJP*): this method is based on a merge join operation performed over the *R-lists* for every fixed predicate in $\mathcal{S}$. The *IJP* uses the *R-lists* for pruning and the *T-lists* for the *variable* binding. This method is the one chosen as comparison to our proposed solution, since it usually achieves better performance for a wide range of different types of queries;
2. *Dynamic Programming Pattern* (*DPP*): this method performs a subsequence matching between every predicate in $\mathcal{S}$ (including *variables*) and the trajectory approximations stored as the *T-lists*. The *DPP* uses mainly the *T-lists* for the subsequence matching and performs an intersection-based filtering with the *R-lists* to find candidate trajectories based on the fixed predicates in $\mathcal{S}$;
3. *Extended-KMP* (*E-KMP*): this method is similar to *DPP*, but uses the Knuth-Morris-Pratt algorithm [16] to find subsequence matches between the trajectory representations and the query pattern;
4. *Extended-NFA* (*E-NFA*): this is an NFA-based approach to deal with all predicates of our proposed language. This method also performs an intersection-based pruning on the *R-lists* to fast prune trajectories that do not satisfy the fixed spatial predicates in $\mathcal{S}$.

4 Proposed FPGA-based hardware implementation

In this section, pattern queries are evaluated in hardware on an FPGA device. As trajectories are compared against hundreds, and potentially thousands, of pattern queries, manually developing custom hardware code becomes an extremely tedious (and error prone) task. Unlike software querying platforms, where a single (or set of) generic kernel can be used for the evaluation of any query pattern, hardware is at an advantage when each query pattern is

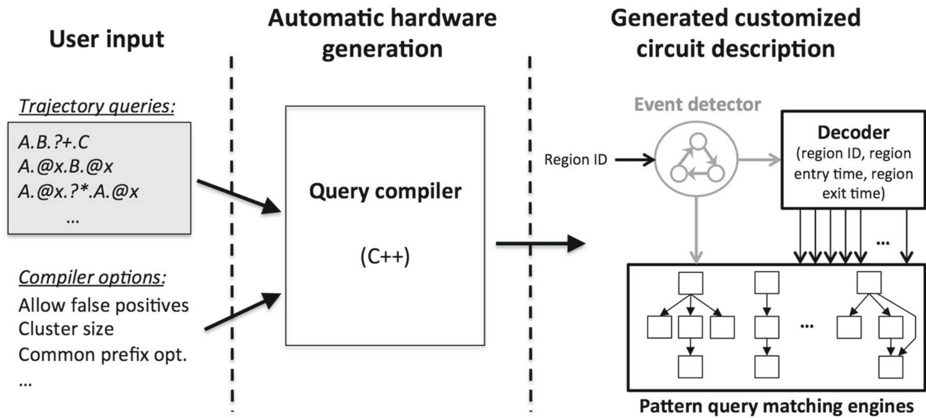


Fig. 2 Query-to-hardware tool flow

mapped to a customized circuit. Customized circuitry has the benefits of only utilizing the needed resources out of all (limited) on-chip resources. Furthermore, the throughput of the query evaluation engines is limited by the operational frequency (hardware clock) which can in-turn be optimized to maximize performance.

For this purpose, a software tool written in C++ was developed from scratch (more than 6,500 lines of code), taking as input a set of user-specified pattern queries $\mathcal{Q}$, and automatically generating a customized Hardware Description Language (HDL) circuit description (see Fig. 2). A set of compiler options can be specified, such as the degree of matching accuracy (reducing/eliminating false positives), and whether to make use of certain resource utilization (common prefix) and performance (clustering) optimizations.

Utilizing a query compiler provides the flexibility of software (ease of expression of queries from a user perspective), and the performance of hardware platforms (higher throughput), while no compromises are introduced.

4.1 High level architecture overview

As depicted in Fig. 2, assuming an input stream of pairs ⟨*location, timestamp*⟩, the first step consists of translating the location onto semantic data; specifically, the region-IDs are of interest, using which the query patterns are expressed. The computational complexity of translating locations to regions depends on the nature of the map, and is discussed below:

1. **Regions defined by a grid map:** in this case, simple arithmetic operations are performed on the locations. These can be performed at wire speed (no stalling) on an FPGA;
2. **Polygon-shaped regions:** in this case, there are several well-defined point-in-polygon algorithms and their respective hardware implementations available (e.g., see [9, 13, 15, 33]). However, none of these can operate at wire speed when the number of polygons is large. Here, the locations of vertices are stored off-chip in carefully designed data structures. The latter are traversed to locate the minimal set of polygons against which to test the presence of the locations.

As the design of an efficient location-to-region-ID block is orthogonal to pattern query matching, in this work a grid map is assumed, and the location-to-region-ID conversion is

abstracted away and computed offline. The input stream to the FPGA consists of ⟨*region-ID, timestamp*⟩ pairs. A high level overview of the generated FPGA-based architecture is depicted at the right-hand side of Fig. 2.

An event detector controller translates the ⟨*region-ID, timestamp*⟩ pairs to ⟨*region-ID, ts-entry, ts-exit*⟩ tuples. The latter are then passed to decoders which transform the region-ID into a one-hot signal (one wire per region ID, s.t. only one wire is high at a time), and evaluate comparisons on entry and exit timestamps as needed by pattern queries. Making use of decoders greatly reduces resource utilization on the FPGA, as computations are centralized and redundancies are eliminated. Note that the decoder is only associated with regions needed by the queries attached to it.

Next, a set of flexible pattern query evaluation engines are deployed, providing performance benefits through the following two parallelization opportunities:

1. **Inter-pattern parallelism:** where the evaluation of all pattern queries is achieved in parallel. This parallelism is available due to the embarrassingly parallel nature of the pattern matching problem;
2. **Intra-pattern parallelism:** where the match states of all nodes within a pattern are evaluated in parallel.

The throughput of pattern query matching engines is limited to one event per cycle. Given the current assumed streaming mechanism, events are less frequent than region-IDs.

Lastly, once a trajectory is done being streamed into the FPGA, the match state of each pattern query is stored in a separate buffer. This in turn allows the match states to be streamed out of the FPGA from the buffer as a new trajectory is queried (streamed in), hence, exploiting one more parallelism opportunity.

A description of the hardware query matching engines follows. While the discussion focuses on predicate evaluation, timing constraints are evaluated in a similar manner in the region-ID decoder, and are hence left-out of the discussion for brevity.

4.2 Evaluating patterns with no variables

We now describe the case of pattern queries with no variables. This approach is borrowed from the NFA-based regular expression evaluation as proposed in [18, 34]. Figure 3a depicts the matching engine respective to the pattern query $A.B.?^*.A$, and Fig. 3b details the matching steps of that query given a stream of region-ID events. Each query node is implemented as:

1. A one-bit buffer, implemented using a flip-flop (Fig. 3b), indicating whether the pattern has matched up to this node. All nodes are updated simultaneously, upon each region-ID event detected at the input stream;
2. Logic preceding this buffer, to update the match state (buffer contents).

As each buffer indicates whether the pattern has matched up to that predicate, a query node can be in a matched state if, and only if:

1. All previous (non-wildstars ?*) predicates up to itself have matched. Wildstars are an exception since they can be skipped by definition (zero or more). To perform this check, it suffices to check the match state of the first previous non-wildstar node (see the node bypass in Fig. 3a);
2. The current event (as noted by the region-ID decoder) relates to the region of that respective node. Wildcards are an exception, since by definition, they are not tied to a

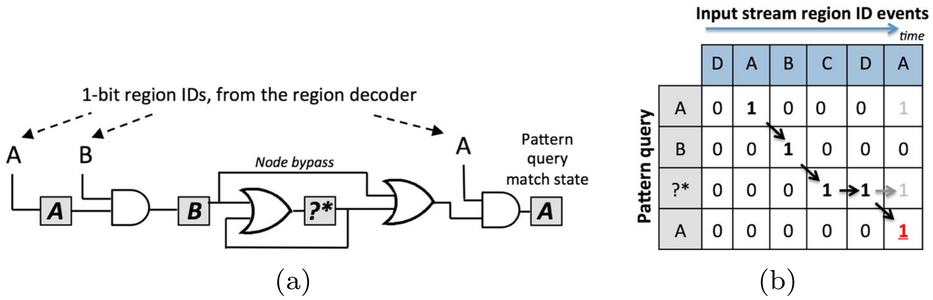


Fig. 3 **a** Query matching engines respective to the pattern query $A.B.?^{*}.A$, and **b** an event-by-event overview of the matching of the query

region-ID. Centralizing the comparisons and making use of a decoder helps considerably reducing the FPGA resource utilization respective to this inter-node logic (see the AND-gates in Fig. 3a). This is in contrast to reading the multi-bit encoded region-ID and performing a comparison locally;

3. It is a wildstar/wildplus ($?^{*}/?^{+}$), and it was in a match state at some point earlier. Wildstar and wildplus are **aggregation** nodes that, once matched, will hold that match state (see the OR-gate prior to the $?^{*}$ node in Fig. 3a).

Looking closer at Fig. 3b, each cell reflects the match state of a query node. All cells in a column are updated in parallel upon an event at the input stream. A '1' in a cell indicates that the query has matched up to that node; for a query to be marked as matched, a '1' should propagate from the first node (top row) to the last node (bottom row). As wildstar (and wildplus) nodes act as aggregators, they hold a *matched* state once activated; hence, a '1' can propagate "horizontally" only at wildstar (and wildplus) nodes. Grey cell contents indicate *matched* states that did not contribute to the detected matched query state in red color, but could contribute to later matches. The '1' depicted in red color in Fig. 3b indicates that the query was detected in the input stream.

4.3 Evaluating patterns with variables and without wildstar/wildplus

Supporting variables in pattern query matching requires an added level of memory saving. The basic rule of variables is that *all instances of a given variable need to match the same region-ID for a variable to be in a match state*. When no aggregator nodes $?^{+}/?^{*}$ are used, the distance between these two region-IDs occurring is the number of nodes between the variable instances in the query.

One possible way for software systems to handle this would be to store, at each variable node (*in a matched state*), all the region-IDs encountered throughout the stream. A post-processing step would carefully intersect, for each variable, all stored region-IDs vectors. While that is a valid approach, storing region-IDs for each variable node of each pattern query is problematic as streams are longer. Furthermore, this is not needed unless aggregator nodes $?^{+}/?^{*}$ occur in between variable occurrences; these cases are detailed in Sections 4.4 and 4.5. As FPGAs allow the deploying of custom matching engines for each pattern, matching pattern queries at streaming (no-stall) mode can be achieved here, with no post processing.

To handle variables in hardware, the first instance of a given variable in a pattern query forwards the event detector's output encoded (multi-bit) region-ID alongside the incoming

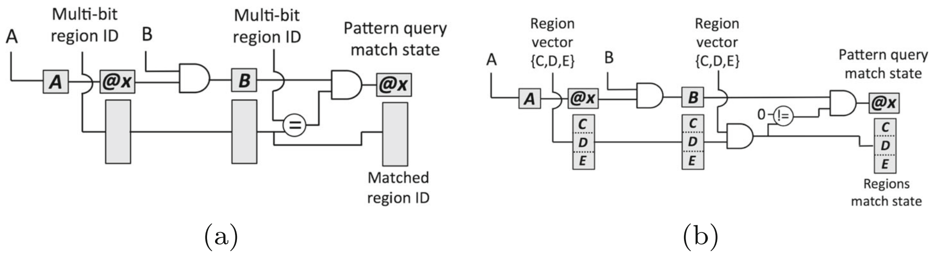


Fig. 4 Query matching engines respective to the pattern query $A.@x.B.@x$, **a** without and **b** with a region set constraint $\{C, D, E\}$ on $@x$

match state (see the second node in Fig. 4a). Some cycles later (depending on the location of variable instances in the pattern), every instance of that variable in the query would match the event detector's region-ID to the forwarded region-ID. If these match, then the region-ID is again forwarded, and the variable instance indicates a *matched* state. Stated in other terms, at a variable node (instance) in a query, a match state is indicated if the current region was encountered earlier (given a fixed implied distance), and all match state propagation checks in between were valid (implying the distance).

Note that an encoded region-ID is used since it is smaller in bit size than a decoded ID, and any region can potentially satisfy the pattern query variable (i.e., variables are essentially a subset of wildcards). Also note that non-variable predicates buffer the forwarded region-ID, though no manipulation of the latter is required. Additionally, one set of region-ID buffers is required per variable, starting from the first occurrence of that variable.

The same solution is applicable to pattern queries containing variables with region sets. Figure 4b shows the matching logic for the pattern $A.@x.B.@x$ where $@x$ is constrained by the regions $\{C, D, E\}$. Here, instead of storing the encoded region-ID in the variable buffers, the latter would hold, for each region in the set, a single bit. At the first occurrence of a variable, the buffer holds a one-hot vector, because input stream events are relative to one region only. Upon later instances of that variable, AND-ing the incoming region set buffer with specific bits of the region-ID decoder output will help indicating for which regions (if any) the pattern matches.

The above approach is similar to replicating the matching engine for each region in the variable region set constraint. For instance, the query in Fig. 4b can be seen as three queries, namely $A.C.B.C$, $A.D.B.D$ and $A.E.B.E$. However, the above approach offers much better scalability when multiple variables are used per pattern: replicating the pattern for each combination of variable regions would result in an exponential increase in resource utilization versus employing the aforementioned style of propagating buffers. Another advantage of the propagating region set variable buffers, when dealing with wildstar/wildplus pattern predicates, is described in the following.

We now describe an alternative "relaxed" implementation of the variable region set constraint, with the goal of saving considerable hardware resources, though at the expense of introducing false positives. Instead of keeping a propagating buffer holding information on each region in the set, the match state can be updated if *any* of the regions in the set are decoded using a simple OR-gate. Figure 5a depicts the gate-level implementation of the query $A.@x.B.@x$ $\{@x : C, D, E\}$, such that the variable region set constraint is implemented as an OR. Thus, history keeping is minimized, as no exact region information is kept per variable. While this mechanism introduces false positives (as described in Fig. 5b), the

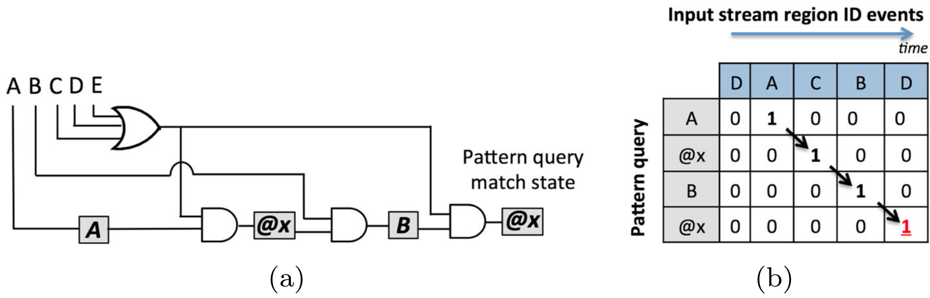


Fig. 5 **a** Query matching engine respective to the pattern query $A.@x.B.@x$ $\{@x : C, D, E\}$, such that the variable region set constraint is implemented as a "relaxed" OR. **b** An event-by-event overview of the matching of the query resulting in a false positive, due to the OR-based implementation of the variable region set constraint

latter can be tolerable depending on the application. Otherwise, a post-processing software step can be performed only on the patterns marked as matched by the FPGA hardware. This approach, however, helps fitting substantially more query engines on the FPGA, a benefit accentuated as the number of variables and the variable region sets' size increase.

4.4 Evaluating patterns with a single variable and with wildstar/wildplus

The remainder of this discussion is applicable to both wildplus and wildstar query nodes. As detailed earlier (Fig. 3a), wildplus nodes act as **aggregator** nodes. When no variables are used, the only propagating information across nodes is a single bit value. In that case, a simple OR gate would suffice for aggregation (state saving).

When a wildplus predicate is located in between two instances of a variable, all values of the region-ID buffer should be stored, and forwarded to the next stages (nodes). Keeping that history is required in order to not result in false negatives. However, due to performance and resource utilization constraints, storing all that history is not desired. Using variable region set constraints, this limitation can be overcome by simply OR-ing the propagating buffer similarly to the match state buffer. This approach would store the information needed, and no history is lost. No false positives are generated, thus pattern evaluation is achieved at streaming mode.

4.5 Evaluating patterns with multiple variables and with wildstar/wildplus

When more than one variable predicate is used in a pattern query, and with wildplus nodes in between instances of both these variables, the previous mechanism can lead to false positive matches, as even more state should be saved than discussed earlier. Figure 6a shows an event-by-event example of a pattern matching resulting in a false positive match. Each cell in the grid holds the values stored inside each respective variable buffer. Buffers for the variable $@x$ are used at each pattern node, whereas buffers for the variable $@y$ span from the second pattern node (i.e. the first $@y$ node), up to the last pattern node.

As described earlier, the wildplus node is the only node in the pattern query allowing horizontal propagation of *matched* states. This is due to the nature of wildplus nodes which hold a *matched* state. As the variable buffers are OR-ed at that wildplus node, they will store the information of the union of all variable buffers encountered at that node. Looking at the $?^+$ row in Fig. 6a, notice that the variable buffers for both $@x$ and $@y$ hold an increasing

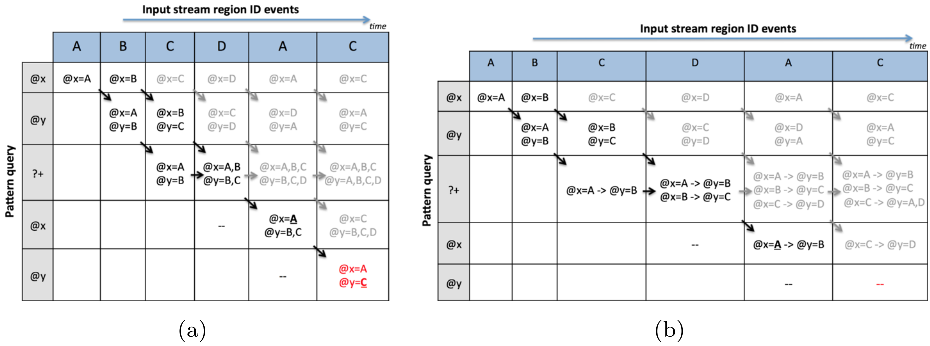


Fig. 6 Event-by-event matching of the pattern query $@x.@y.?^{+}.@x.@y$ $\{@x : A, B, C, D\}$ $\{@y : A, B, C, D\}$. The resulting match in **a** is a false positive; whereas enough state is saved in **b** at the aggregator node ($?^{+}$) to eliminate that false positive

number of regions. That level of stored information is not sufficient, as it will be shortly shown to result in a false positive.

Upon the D event, both variable buffers did not propagate to the second instance of $@x$. That is because the $@x$ variable buffer does not reflect that the previous instance of $@x$ held the value of D (yet). However, on the next event A, the variable buffers propagated, and the $@x$ variable buffer was masked with the event region. Hence, B was removed from the $@x$ variable buffer. The $@y$ variable buffer remains unmodified, since the $@x$ node is not allowed to modify it.

Finally, at the last event C, focusing at the second instance of $@y$ (i.e. the last pattern predicate), a match is shown for $@x{=}A$ and $@y{=}C$. While @x and @y did hold these values at some point, looking closer at the input stream, A and C were initially separated by B, though the query requires that the distance between $@x$ and $@y$ is 1 (back-to-back regions visited).

In order to not result in false positives, the level of history kept at the aggregator node has to be increased. Instead of only storing the union of all variable buffers, the information at the wildplus node should be the set of all variable buffers encountered. To reduce storage, that solution can be simplified such that, for each $@x$ variable value, a list of all corresponding $@y$ values are stored (as shown in Fig. 6b). Focusing on the aggregator row, every value of $@x$ is associated with a list of $@y$ values. These can be deduced from the propagating variable buffers into the wildplus node. Note that $@x{=}A$ is associated with $@y{=}B$. Therefore, the tuple $@x{=}A$, $@y{=}C$ cannot result in a match, as is the case in Fig. 6a.

Nonetheless, implementing this solution in hardware is extremely costly in terms of resource utilization (and impact on the critical path/performance), especially with larger region sets and many variables per pattern. Furthermore, this solution does not scale with many variables, and does not hold with more aggregator nodes.

Another approach to eliminate false positives in such cases is a brute-force implementation of each query using all variable region-set combinations. For instance, the query $\mathcal{S} = @x.@y.?^{+}.@x.@y$ $\{@x : A, B\}\{@y : C, D\}$ can be implemented as four simpler queries, namely: $\mathcal{S}_1 = A.C.?^{+}.A.C$; $\mathcal{S}_2 = A.D.?^{+}.A.D$; $\mathcal{S}_3 = B.C.?^{+}.B.C$; and $\mathcal{S}_4 = B.D.?^{+}.B.D$.

This approach is encouraging when the number of variables and the size of the region sets is relatively small. Otherwise, the implied resource utilization increases too much, even though each query is built using simple matching engines (no propagating variable buffers).

Nonetheless, the common prefix (among similar pattern queries) optimization helps with the scalability.

In order to better evaluate the benefits of each of the above approaches, a study on the resulting false positives versus resource utilization is performed in Section 6. In summary, when pattern queries make use of two or more variables, and with an aggregator node in between the occurrences of these variables, the proposed approaches are:

1. **Making use of propagating variable buffers:** this approach results in the least false positives;
2. **Implementing region set constraints as an OR:** the number of false positives here is a superset of the above case, and resource utilization is minimal. False positives are a superset, since the condition (OR check) to allow a match to propagate through a variable node is a superset of the first approach's variable node conditions (propagating buffers);
3. **A brute-force mapping approach:** this approach map each query as the combination of all variable region-sets. It has **no false positives**, but does not scale well with more variables and larger region sets.

5 Proposed GPU-based implementation

This section describes the mechanisms and optimizations used to effectively evaluate flexible pattern queries on GPUs, based on the three aforementioned approaches: *variable as OR*, *propagating buffers*, *all combinations*. The following study compares the effects of several factors on the performance of GPU implementation, in contrast to its FPGA counterpart.

5.1 GPU solution overview

For the remainder of this study, we use of the Nvidia CUDA terminology [27]. In CUDA, every GPU accelerated application consists of two main parts: the *host code*, which is executed on the general-purpose CPU connected to the GPU via PCIe; and the *GPU kernel code*, multiple instances of which are executed in parallel on graphical processors. The number of kernel instances, i.e., threads, is not limited to the physical number of computing cores on the GPU.

Typical GPU architectures consist of a large number of (simple) compute cores, called Streaming Processors (SPs). SPs are clustered into Streaming Multiprocessors (SMs). Each SM is coupled with small and fast memory buffers used for caching read-only data or intra-SP communication. Furthermore, every instruction fetched gets executed on each SP, which enables GPU to process data in a SIMD (Single Instruction, Multiple Data) manner. Finally all SMs are connected to a high-latency global memory, the latter being the point of communication with the host CPU.

The user specifies the number of kernels to be grouped in *thread blocks*, such that all threads in a thread block execute on a single SM (and can communicate through the shared low-latency memory). In order to further maximize utilization, several thread blocks can be executing simultaneously on the same SM, such that thread block scheduling is automated by the CUDA framework.

In our GPU solution, the event detector (Fig. 2) is implemented as part of the host code, converting streams of ⟨*location, timestamp*⟩ pairs into ⟨*region-ID, ts-entry, ts-exit*⟩ tuples.

Three GPU kernels are implemented, each respective to one of the three query matching approaches: *variable as OR*, *propagating buffers*, and *all combinations*. Kernels take as input pattern query definitions along with compiler options (Section 4). These parameters enable specific optimizations (e.g., common prefix, pattern minimization), as well as manage performance specific characteristics (e.g., cluster size). Each kernel processes each event from the stream passed from the host code.

Our GPU-based solution leverages thread scheduling and grouping techniques to attain the same levels of parallelism as in the FPGA-based solution:

1. **Inter-pattern parallelism:** all pattern queries are evaluated in parallel, by scheduling their thread blocks on different SMs, or sharing a single SM in a time multiplexed manner;
2. **Intra-pattern parallelism:** individual predicates within a pattern are evaluated in parallel on different SPs, when the thread block is executed.

In the following subsections we provide a detailed description of the proposed algorithms implementation on GPUs, as well as the parameter encoding scheme and the applied GPU-specific optimizations.

5.2 GPU kernel personality

One of the key benefits of the CUDA platform lies in its ability to scale: developer needs to implement only a single GPU kernel, the execution framework will spawns multiple copies of the kernel in parallel threads. Given the fact that the same kernel code is used to evaluate every predicate in the pattern, all predicate-specific information should be passed to the kernel as an input parameter. The configuration, referred to as *personality*, dictates all the predicate's properties, which are needed to carry out individual predicate matching.

The information encoded in this personality is depicted in Table 1. Some of the encoded fields (e.g., *type*, *timestamps*) have the same meaning for every predicate. Field *ID* is defined for each predicate, but its semantics depend on the predicate's type. Finally, a set of type-specific fields is used (e.g., *isFirstOccurence* for variables, *prevPredicate* and *prevPrevPredicate* for wildcards, *isLast* and *matchId* for last predicates in the pattern).

Given an input query set (and optimization parameters), the host constructs all correspondent personalities. Kernel personality information is compacted and then transferred, before kernel execution, to the GPU's global memory. Kernels then read their personalities

Table 1 GPU Kernel personality storage format

Field	Description	Size
type	Predicate type: *fixed*, *wildcard*, *wildplus*, *variable*	2 bits
isLast	**true** for last predicates in the pattern, **false** otherwise	1 bit
isFirstOccurence	**true** for the first occurrence of variable, **false** otherwise	1 bit
timestamp from	Beginning of predicate's temporal interval	4 bytes
timestamp to	End of predicate's temporal interval	4 bytes
ID	ID of the region (for fixed predicates), or variableId	2 bytes
prevPredicate	Pointer to the previous predicate in the pattern	2 bytes
prevPrevPredicate	Pointer to the predicate in the pattern, before previous	2 bytes
matchId	Unique match ID for the last predicate in the pattern	2 bytes

once in the beginning of execution, and store the configuration for later usage to alleviate GPU global memory fetching. Personality fields are placed in registers to insure the fastest possible access.

Algorithm 1 Propagating Buffers Kernel

```
1: level ← 0, prevCol ← 0, matchNum ← 0
2: for all region events in trajectory stream do
3:     col ← level % 2
4:     if prevPredicate ≠ null then
5:         buffers[row][col].regions ← buffers[prevPredicate][prevCol].regions
6:         if type = wildstar or type = wildplus then
7:             if type = wildstar then
8:                 buffers[row][col].regions ← buffers[prevPrevPredicate][prevCol].regions
9:             buffers[row][col].regions ← buffers[row][prevCol].regions
10:    match ← buffers[prevPredicate][prevCol]
11:    switch type do
12:        case fixed:
13:            match ← match and (id = region.id)
14:        case wildstar:
15:        case wildplus:
16:            match ← match or buffers[row][prevCol].match
17:        case variable:
18:            varRegion ← buffers[row][col].regions[id] ∩ region.id
19:            if varRegion = ∅ then
20:                match ← false
21:            else
22:                if varRegion is not initialized then
23:                    buffers[row][col].regions[id] ← region
24:                else
25:                    if isFirstOccurence then
26:                        buffers[row][col].regions[id] ← region
27:                    else
28:                        match ← match and (buffers[row][col].regions[id] ∩ region)
29:    if isLast and match then
30:        output[matchNum++] ← matchId
31:    prevCol ← col
32:    level++
```

5.3 Trajectory querying kernel

For the GPU kernel implementation, we use the same dynamic programming-based solution described in Sections 4.2-4.5. Algorithm 1 represents a simplified version of the *propagating buffer* query processing approach. Details of the GPU kernels for the other two approaches (*variable as OR* and *all combinations*) are omitted for brevity, but can be simply derived from the aforementioned algorithm. Note that the variable termed *row* refers to thread ID, which is unique within the block (in CUDA it is predefined as variable *threadIdx*). Similarly *prevPredicate* (and *prevPrevPredicate*) represents the ID of the thread, executing the predicate's prefix.

Algorithm 1 describes a sequence of operations, specific for a single predicate within a query pattern, executed in a separate thread. To process a query pattern as a single unit, individual threads should be grouped into thread blocks. Individual threads within a block communicate through shared memory. For most use cases, all predicates of the pattern query fit into a single thread block.

In our algorithm, the *buffers* array, residing in shared memory, is used to pass the matching state between predicates. Each predicate saves its state in a *buffers* array entry, which

includes the *propagating buffer regions* (used to save the variable's state) and a boolean value *match* (used to carry the match state from one predicate to another). In case of a variable predicate coupled with a region set, its *propagating buffer regions* are pre-populated on the host side.

Note that in the dynamic programming problem we are solving, only the most recent *buffers* state is of interest to our purposes; hence, all previous states of the *buffers* array can be overwritten with the new ones. As a result, the memory allocated for the *buffers* array is bounded to two columns (one column for the current and one for the previous state) and N rows, where N is the size of thread block. Each kernel loops through every region event in the trajectory event stream and performs some computations on the current column. After all computations are done, the current column is swapped between 0 and 1 (and from 1 to 0 on the next iteration) on line 3.

Lines 4-9 are copying and merging (in case of wildcard nodes) contents of *propagating buffers* of the node's parent with the *propagating buffers* of the current node. Lines 11-28 contain the main matching logic: initially the match is propagated from the node's parent, followed by node-specific matching restrictions. In the case of a fixed-region predicate (line 13), a match is propagated further only if the predicate's region-ID is the same as the current trajectory event region. If the pattern is a wildcard, a match can be propagated from the same node as well as its parent (line 16).

The variable predicate (lines 17-28) is the more interesting case. Depending on the variable region set, a number of situations can occur:

– The region set is filled, but the current trajectory event region does not overlap with the set (line 20). This will result in a failed pattern match;
– The region set is initially empty (line 23). In this case the current trajectory event region should be recorded into the *propagating buffer regions*;
– The region set is filled, and the current trajectory event region overlaps with the set (lines 25-28). If this is the first occurrence of a variable in the pattern, the trajectory event region is recorded in the *propagating buffer regions* (line 26); otherwise, the regions which caused the previous variable's matches should be compared to the current trajectory event region (line 28).

Finally, the match state of the whole pattern is reported on line 30.

5.4 Performance optimizations

We now describe optimizations applied to maximize the GPU utilization and to carefully utilize the memory hierarchy, with the goal of boosting performance.

Due to architecture limitations not all the threads in a block are executed simultaneously. Threads are executed in groups of 32, computing in a SIMD fashion. Each such group of threads is referred to as a *warp*. Scheduling on SMs takes place at the warp level, rather than the thread block level, which allows to co-allocate warps from different blocks on the same multiprocessor. The GPU hardware has a limit on the number of warps that can be supported without saturating all hardware resources. The ratio between the number of active warps and this limit is called the *GPU occupancy*, which describes the load on the GPU-cores. It is desirable to have high occupancy, because more active warps can better mask memory access latency.

The number of the active warps depends on the amount of shared memory and the number of registers consumed by a warp. Both shared memory and registers are allocated on per-SM basis and shared between all warps, which are executing on this SM. Since a

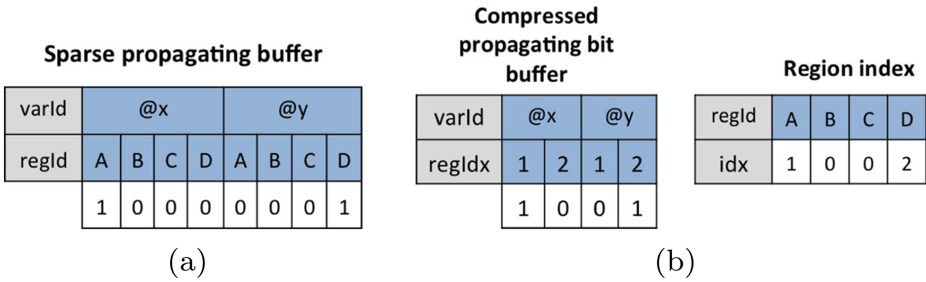


Fig. 7 *Propagating buffer* compression on GPU: **a** with the sparse version, and **b** applying indexing to obtain compressed buffer

particular kernel implementation dictates the number of registers used by a thread (hence the warp) our first optimization aims at reducing the amount of shared memory used by a single thread.

In Algorithm 1, only the *buffers* array is stored in the shared memory. The length of this array is proportional to the number of threads in the block, the number of variables and the size of *propagating buffer*. In the worst case, the length of the *propagating buffer* should be the same as the number of unique regions in the trajectory event stream to be able to record all of them.

In order to reduce shared memory contention, we have to eliminate sparse *propagating buffers* with a compressed version using the indexing technique depicted in Fig. 7. The un-optimized version in (a) shows that the *propagating buffer*, indexed by region-ID, may contain a lot of 0 entries, because the trajectory may span only a small part of the grid. To overcome this, we introduce an additional thread block-specific region index, which indexes only the regions visited by a trajectory (or used in region sets of variables), depicted in Fig. 7b. Although the index itself is a sparse array, it is stored only once for all threads in a block, as opposed to having a sparse un-indexed array per thread. Moreover, as a *propagating buffer* is used to record whether a variable had a specific region value, it could be compressed down to a single bit.

A similar technique is used to index variables. Since a propagating buffer needs to be stored for each variable in a pattern, a naïve way would be to store the variable's propagating buffer in an array, indexed by the block-specific variable ID. However propagating buffers need to record matches of variables encountered only within a pattern. The number of variables in the pattern for all practical purposes is usually significantly smaller than the total number of unique variables in the thread block. Thus we introduce an index that assigns a pattern-unique ID to every variable in the cluster.

Note that these indexing optimizations incur additional preprocessing time and introduce new parameters (indexes) passed to the GPU kernel. The described index needs to be recalculated from scratch for each trajectory, since the maximum number of unique regions (which determines the length of the index) is different for each trajectory event list. Therefore, pattern querying on GPU could be done in streaming mode only within a single trajectory. GPU implementation could be thought of as an intermediate approach between the software version, which needs a dedicated index creation preprocessing step for both trajectories and event regions, and the FPGA solution, which is capable of processing everything in streaming mode.

Unlike the fast and small shared memory, used for intra-thread communication GPU, the *global* memory is mainly used to transfer initial parameters to the GPU kernel, or to store

trajectory data. This feature comes at a certain price: latency of global memory is several orders of magnitude higher, compared to shared memory. Although it is not possible to eliminate all global memory accesses, we applied *coalescing* to retrieve/update data from/to global memory. Reading/writing data in such manner means that individual reads/writes from the threads within a warp could be combined into a single operation over a contiguous memory block. To implement the coalescing optimization, multidimensional arrays, which is a storing format for kernel parameters, are indexed in a way that uses *threadIdx* as the “outermost” index field.

6 Experimental evaluation

We now present an extensive experimental evaluation of both proposed FPGA- and GPU-based solutions. We first describe the datasets used in the experiments, followed by the experimental setup. We then detail a thorough design space exploration on the proposed architecture, alongside with a study on matching accuracy. Finally, we show the performance evaluation between hardware FPGA architecture, parallel GPU solution and the CPU-based software approach.

While users typically make use of mixes of query characteristics (query length, number of variables, use of wildcards, etc), experiments are carried out on isolated query characteristics in order to better study their respective effects.

6.1 Dataset description

In our experimental evaluation, we use four real trajectory datasets. The first two datasets are the *Trucks* and *Buses* from [7]. Both datasets represent moving objects in the metropolitan area of Athens, Greece. The *Trucks* dataset has 276 trajectories of 50 trucks where the longest trajectory timestamp is 13,540 time units. The *Buses* dataset has 145 trajectories of school buses with maximum timestamp 992. The third dataset, *CabsSF*, consists of GPS coordinates of 483 taxi cabs operating in the San Francisco area [30] collected over a period of almost a month. The fourth dataset, *GeoLife*, contains GPS trajectory data generated from people that participated in the GeoLife project [41] during a period of over three years. This dataset has 17,621 trajectories with total distance of ≈1.2 million Km and duration of ≈48,000 hours.

6.2 Experimental setup

For simplicity of the experimental evaluation, we partition the spatial domain in uniform grid sizes. These grid cells become the alphabet for our pattern queries. To generate relevant pattern queries for each dataset, we randomly sample and fragment the original trajectories using a custom trajectory query generator. The length and location of each fragment are randomly chosen. These fragments are then concatenated to create a pattern query. We generate up to 2,048 pattern queries with different number of predicates, variables, and wildcards. The location of variable/wildcard in the query is randomly chosen.

Our FPGA platform consists of a Pico M-501 board connected to an Intel Xeon processor via 8 lanes of PCI-e Gen. 2 [29]. We make use of one Xilinx Virtex 6 FPGA LX240T, a low to mid-size FPGA relative to modern standards. The PCIe hardware interface and software drivers are provided as part of the Pico framework. The hardware engines communicate with the input and output PCIe interfaces through one stream each way, with dual-clock

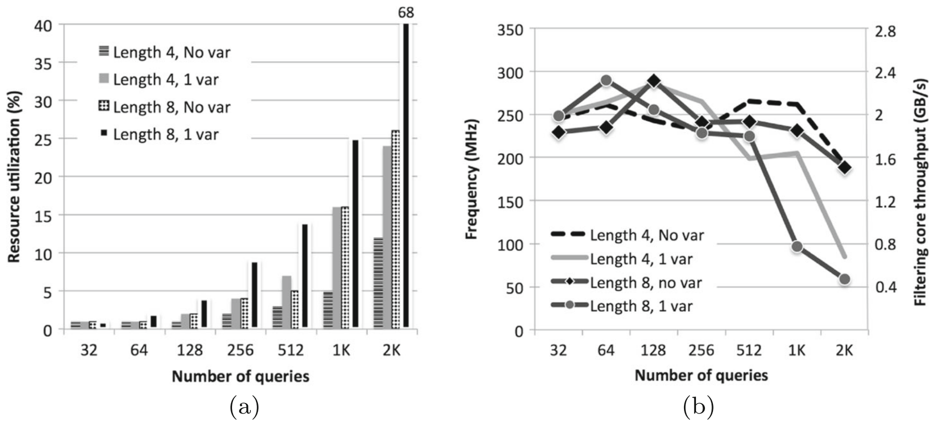


Fig. 8 **a** Resource utilization and **b** respective frequencies/throughput of the FPGA querying engines, such that the number of queries is doubled, the query length is doubled

BRAM FIFOs in between our logic and the interfaces. Hence, the clock of the filtering engine is independent of the global clock. The PCIe interfaces incur an overhead of ≈8% of available FPGA resources. The RAM on the FPGA board is not residing in the same virtual address space of the CPU RAM. Data is streamed from the CPU RAM to the FPGA. Since the proposed solution does not require memory offloading, RAM on the FPGA board is not used. Xilinx ISE 14 is used for synthesis and place-and-route. Default settings are set.

Our GPU workstation is equipped with a high-end Nvidia Tesla K20 graphic card which has 13 SMs on the board, each consisting of 192 SPs (i.e., 2,496 compute cores in total). The communication between the GPU RAM and main memory always goes through a PCIe interface.

CPU experiments were ran on an Intel Xeon processor running @3.3GHz, attached to 256GB of main memory.

6.3 Design space exploration

6.3.1 FPGA-based approach results

Here we discuss resource utilization and performance (throughput) achieved by FPGA-based querying engines. Figure 8a shows resource utilization. Figure 8b depicts the respective frequencies of the hardware engines while varying **(1)** the number of queries (32, 64, 128, ... 2,048 queries), **(2)** the query length (4 and 8 predicates), and **(3)** the number of variables in a pattern query (0 and 1 variable, with a variable region set of 5 regions).

As the query compiler applies the common prefix optimization, and further resource sharing techniques are exercised by the synthesis/place-and-route tools, resource utilization does not double as the number of queries is doubled. Rather, a penalty of approximately 70% occurs. Similarly, as the query length is doubled, an average increase of 80% in resources is found. However, adding one variable to each query results in, on average, doubling resource utilization. Note that the *propagating buffer* approach is employed for variable matching, and that these buffers propagate from the first occurrence of the variable to the last. Overall, up to several thousands of query matching engines can fit on the target Xilinx V6LX240T FPGA, a mid- to low-size FPGA.

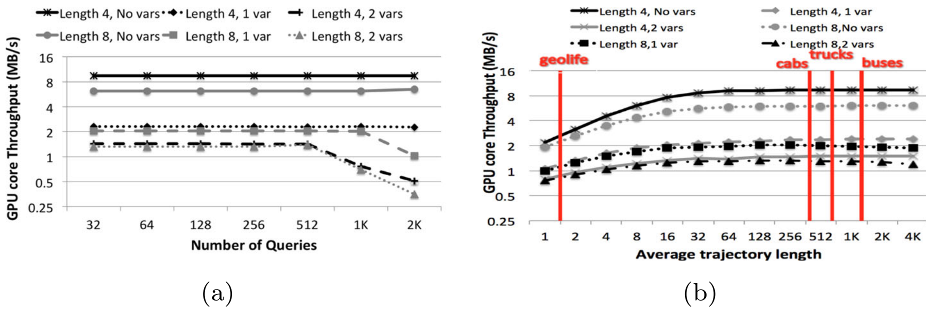


Fig. 9 GPU core throughput for variable query length and number of variables as a function of **a** number of queries and **b** average trajectory length

While the aforementioned numbers address FPGA scalability of the proposed matching engines, Fig. 8b details the respective achievable performance in terms of:

1. **Operational frequency (MHz)**: measured as a function of the critical path, i.e., the longest wire connection of the FPGA circuit. This number is obtained post the place-and-route process of the FPGA tools;
2. **Throughput (GB/s)**: as the query matching engines process one ⟨*region-ID, timestamp*⟩ pair per hardware cycle, the FPGA throughput can be deduced from the circuit's operational frequency, given that the size of each input pair is 8 Bytes (2 integers). Nonetheless, this computed throughput is respective to the FPGA circuitry, and might not reflect the end-to-end (CPU-FPGA and back) performance, which is platform dependent. The end-to-end measurements are discussed in the sequence.

As the number of queries increases, frequency/throughput is initially around the 250MHz/2GBs mark. Fluctuations are due to the heuristic-based nature of the FPGA tools, though generally a trend is deduced. As the number of queries becomes too large, frequency drops considerably for queries with variables. The drop is not as steep for queries with no variables; the reason being that queries with variables can be thought of as longer queries (due to the *propagating buffers*). This drop in frequency occurs because of the large fan-out from the *region-ID* decoder to the many sinks, being the query nodes and *propagating buffers*.

Replicating the *region-ID* decoder (and event detector) helps reducing fan-out, and will potentially eliminate it. Each *region-ID* decoder is then connected to a set of queries. We refer to a *region-ID* decoder and its connected queries as a **cluster**. Note that each query belongs to exactly one cluster. The query compiler is developed to take as input parameter the cluster size, as a function of query nodes. Thorough experimentation shows that clusters need not hold less than 1,024 or even 512 query nodes (data omitted due to lack of space). Larger clusters result in performance deterioration; smaller clusters do not offer any benefits in performance, rather present an increase in resource utilization (due to the replication of the *region-ID* decoder and event detector per cluster).

6.3.2 GPU-based approach results

Performance results achieved with our GPU-based implementation are depicted on Fig. 9. The set of queries used in this experiment is a superset to the FPGA counterpart: we also include patterns of length 4 and 8 with 2 variables. Figure 9a shows that the GPU matching

throughput for queries without variables is almost constant, which signals that the GPU is underutilized. Doubling the query length (without variables) causes the throughput to decrease less than twice, due to the common prefix optimization. Query length has the same effect on GPU throughput as it has on FPGA resource utilization.

There is also a huge performance gap between the queries with no variables and queries with at least 1 variable; this is the penalty paid for storing *propagating buffers* in the GPU shared memory. This penalty is prominent when adding the first variable. Adding more variables to the queries increases the *propagating buffer* size by only a constant value, hence the throughput is decreased less. This also shows the effectiveness of the indexing optimization applied to reduce the shared memory contention.

When considering queries with at least 1 variable, we observe a “breaking point” where the throughput drops rapidly (as the number of queries increases; in this figure the number of queries keeps doubling from 32 to 2048). For example, for queries with 2 variables the breaking point appears between 512 and 1024 queries. This point, indicates that all available GPU computing cores have been used. After that the GPU becomes over-utilized instead of been underutilized, and any further increase in the number of queries will result in their linear execution. Long queries with considerable number of variables tend to use more GPU resources, hence “breaking point” appears “earlier” in the graph.

We also experimented with other structural characteristics of the queries, namely, the wildcard predicate probability and the size of the variable’s region set (experiments not shown due to lack of space). Changing the wildcard predicate probability did not have any observable effect on the GPU throughput. We observed that GPU throughput for queries with an empty variable region set is on average 2 times lower than queries containing variables with non-empty region sets (irrespectively of the set size). This is because, for a variable predicate without any region constraints we need to record all possible matches in its own propagating buffer; this incurs additional memory loads. In contrast, for variables with non-empty region sets the propagating buffer already contains the fixed-sized region set.

Figure 9b GPU explores throughput as a function of average length, calculated for synthetically generated datasets. Peak throughput can be achieved only for the trajectories with average length roughly between 64 and 128 regions. For smaller regions execution time is nominated by GPU kernel launch overhead. Average lengths of the real life datasets, used later in performance evaluation, in shown on the same figure.

The GPU implementation also uses a notion of cluster like the FPGA counterpart. However, since all region detection happens on the host side, the GPU cluster is essentially defined as a set of patterns, evaluated together in parallel, i.e. a thread block. To determine the optimal block size we carried out a series of experiments measuring the performance for different block sizes. For these experiments, we used queries of length 5 with 2 variables and variable region set of size 10. The block size ranged from 32 to 1,024 threads, which is the maximum block size along one dimension in the CUDA architecture. Figure 10 depicts that thread block of size 128 yields the best performance.

6.4 Query engine implementations and false positives

As described in previous sections, a query holding variables can be evaluated in one of three ways, namely:

1. **Variable as OR:** implementing the region set constraints as *ORs* (resulting in most false positives);

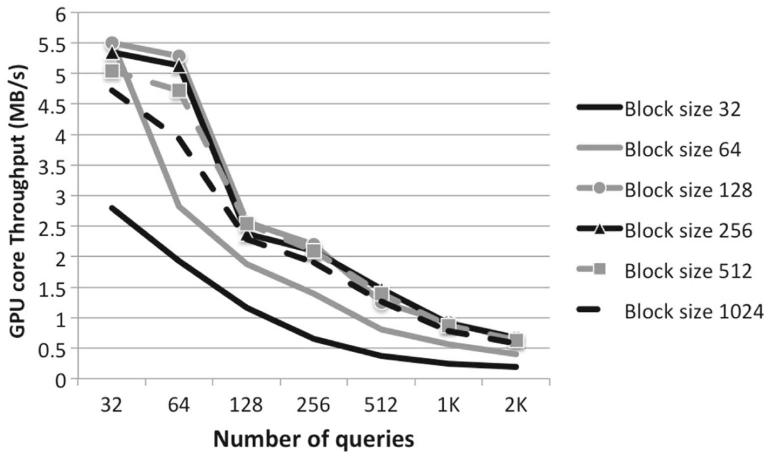


Fig. 10 GPU throughput, with respect to block size/queries

2. **Propagating buffer:** making use of *propagating buffers* (false positives arise only when using multiple variables alongside wildstar/wildplus nodes);
3. **All combinations:** brute-force mapping of each query as the combination of all variable region sets (no false positives).

Figure 11 shows the experiment, in which the number of false positive matches for each of the three previously discussed query engine implementations was evaluated. In this study the matching accuracy is recorded for each implementation of 100 long queries, over three datasets, namely *Trucks*, *Buses* and *CabsSF* (the results for the *GeoLife* dataset follow the same pattern). The matching accuracy is measured as $(100 - (\textit{percentage of false positives}))$ over all trajectories in each respective dataset, where higher reflects better accuracy. Queries are generated using our query generator tool, where each query contains two variables, as well as one or more aggregator ($?^*/?^+$) nodes. Note that the *Propagating buffers* approach does not result in any false positives, unless multiple variables are used alongside aggregators.

As expected, the *All combinations* approach results in no false positives. However, while the *Variable as OR* technique results in the most false positives (as expected), the matching

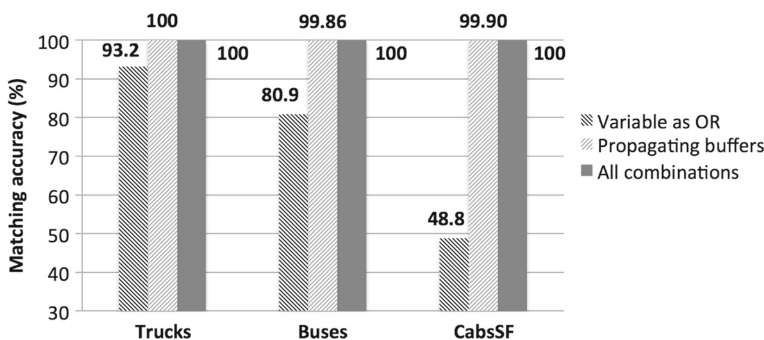


Fig. 11 Matching accuracy of 100 long queries, measured as $(100 - (\textit{percentage of false positives}))$ over all trajectories in each respective dataset. Higher reflects a better accuracy

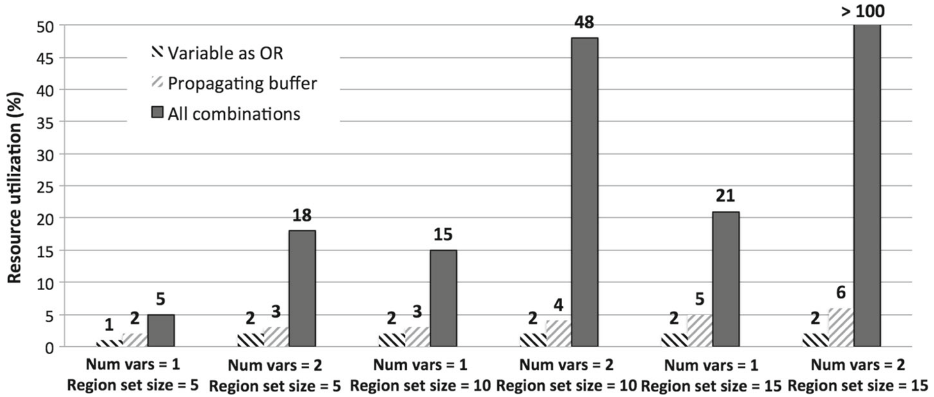


Fig. 12 Scalability of the FPGA-based solution for the three implementations over 100 queries of length 6 with variables

accuracy varies from high (93.2%), to a somewhat low (48.8%). On the other hand, matching accuracy is close to perfect (> 99.8%) for the *Propagating buffers* implementation, even as false positives increase as a result of the *Variable as OR* implementation. No false positives are recorded on the *Trucks* dataset when making use of the *propagating buffers*.

6.4.1 FPGA-based engine implementation scalability

The following study presents scalability of each of three aforementioned approaches as the number of variables and the region set size are increased. Figure 12 illustrates the FPGA resource utilization of 100 queries of length 6 holding variables, implemented in each of the aforementioned three approaches. The varied factors are the number of variables in each pattern query, and the respective region set size.

When implementing a *variable* as *OR*, each variable node is replaced with a simpler *OR* node. Thus, as expected (see Fig. 12), increasing the number of variables has almost no effect on resource utilization. The same applies to increasing the region set size. On the other hand, the *propagating buffer* technique starts off as utilizing slightly less than double the resources of the *variable* as the *OR* approach. Furthermore, doubling the region set size results in a 50% area penalty. Doubling the number of variables per pattern query exhibits similar behavior.

Finally, when transforming a query into a set of queries based on *all combinations* of the region sets, resource utilization starts off as more than double that of the *propagating buffer* technique. Doubling the number of variables naturally has a steeper effect than doubling the region set size on resource utilization. Note that the common prefix optimization helps with the scalability of this approach. Nonetheless, when using two variables with region set size of 15, the resulting circuitry did not fit on the FPGA. Practically, it is best to make use of this approach for critical pattern queries where false positives are not tolerated.

While the mileage of the *Variable as OR* implementation may vary, its scalability is key. Even when false positives are not tolerable, query matching engines can employ this technique, where the FPGA would be used as a pre-processing step with the goal of reducing the query set. The same applies for the *propagating buffers* implementation technique, where the query set would be reduced the most. Since the performance of CPU-based software approaches scales linearly with the number of pattern queries, reducing the query set has

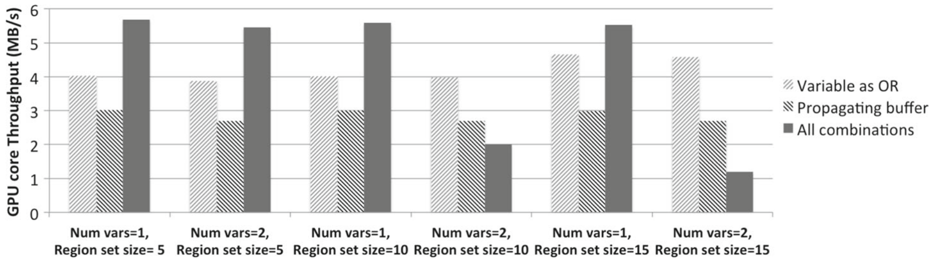


Fig. 13 Scalability of GPU-based solution for 128 queries of length 8 with variables

desirable advantages, especially that the time required for this pre-processing FPGA step is negligible.

6.4.2 *GPU-based engine implementation scalability*

Figure 13 depicts the results from the scalability experiments carried out on the GPU. Here the considered measure is throughput. All experiments were using 128 queries between 1 or 2 variables, while the variable region set size was growing from 5 to 15. For both *variable as OR* and *propagating buffers* approaches, the number of variables does not significantly affect performance. This could be explained by the fact that the "breaking point" has not yet been reached. On the contrary, the throughput of *all combinations* shows a significant drop as the variable region set size and the number of variables grow, since the overwhelming number of generated queries quickly occupies all available GPU computing cores.

6.5 Performance evaluation

In the last set of experiments, we compare the performance evaluation between proposed parallel FPGA- and GPU-based solutions and the CPU-based software approach. Figure 14 shows the end-to-end (CPU-RAM to FPGA and back) FPGA throughput of length 4 queries with 1 variable. Throughput is lower from the FPGA filtering core for smaller trajectory files since steady state is not reached, and communication setup penalty is not hidden. For

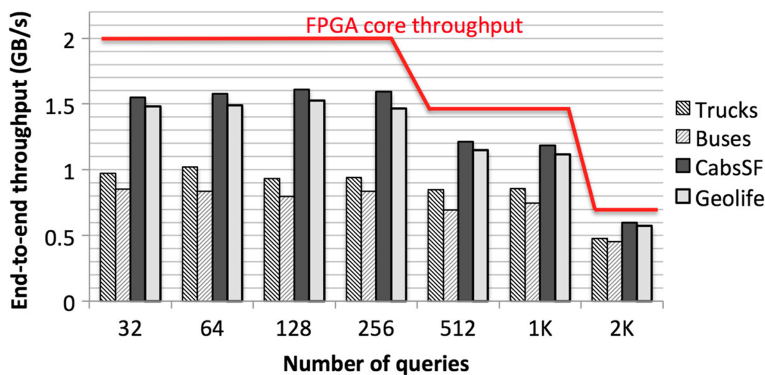


Fig. 14 End-to-end (CPU-RAM to FPGA and back) throughput of queries of length 4 with 1 variable. The throughput of the FPGA filtering core is drawn in red line

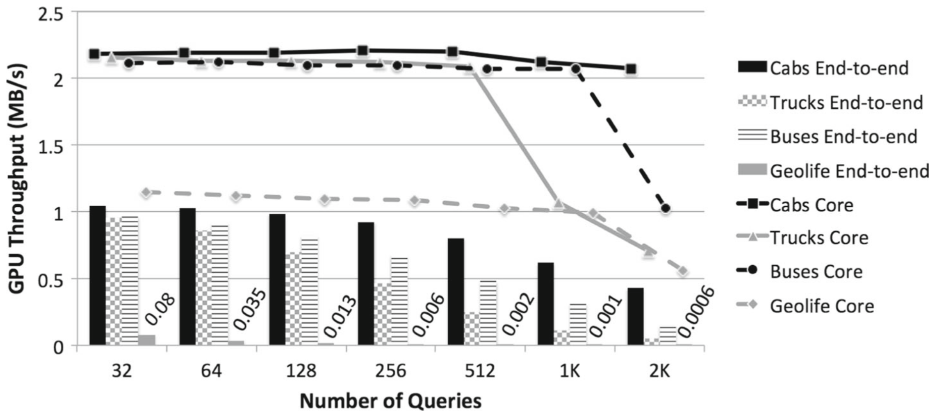


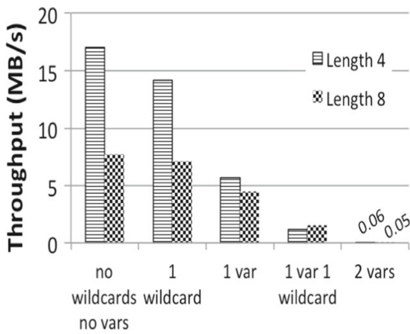
Fig. 15 GPU core and end-to-end throughput for queries of length 8 with 2 variables

larger files, throughput is closer to the FPGA core's, given the physical limitations. Note that the throughput of the FPGA setup is independent of the trajectory file contents, as well as query structure (given a certain operational circuit frequency).

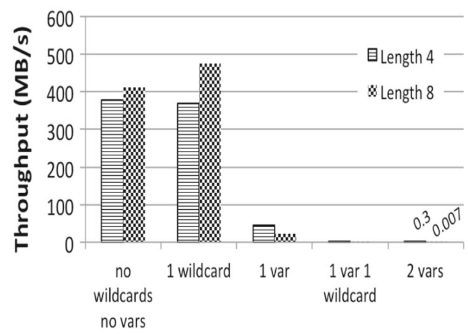
Figure 15 shows GPU core and end-to-end throughput for queries of length 4 with 1 variable. We observed previously (Fig. 9a) that such complex queries eventually will occupy all available GPU computational cores. We see a similar behavior here. The core throughput begins decreasing roughly by a factor of 2 when more than 512 (Trucks dataset) or 1024 (Buses and Geolife) queries are submitted, which means that the GPU throughput reached the "breaking point". Note all datasets, except for *GeoLife*, have the same peak GPU core throughput. As it was showed earlier in Fig. 9b, the GeoLife dataset on average has very short trajectories, which prevents GPU from achieving peak core throughput. End-to-end throughput has much more variance across datasets, and that is due to the pre-processing step taking place on the host CPU. The pre-processing step is susceptible to dataset properties such as the number of trajectories (further exploration into optimizing the pre-processing step is omitted for brevity).

Figure 16 depicts the FlexTrack (software) *IJP* throughput (MB/s) resulting from matching for 2,048 queries with varying properties on the Fig. 16a *Trucks*, Fig. 16b *GeoLife* and Fig. 16c *Buses* datasets. Pre-processing (index building) time is excluded. The current implementation of (CPU-based) FlexTrack is a single-threaded one. One can assume ideal performance scaling with multi-threading enabled; though that assumption is to some extent unrealistic (due to sw threading overhead, NUMA architectures, application not being easily massively parallelizable, etc), it provides an upper bound on the CPU-based performance.

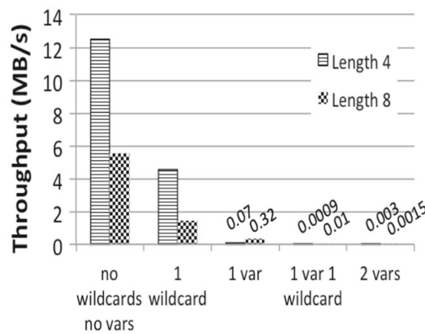
When considering simple queries, throughput is initially higher for the larger dataset (*GeoLife*), where processing steady-state is reached. Throughput drops rapidly as more variables and wildcards are used. Note that software throughput is greatly affected by the query complexity and dataset contents, because intermediate results produced during join operations grow rapidly. On the other hand, both FPGA- and GPU-based implementations are not that sensible to query structure. These implementations are able to achieve considerable speedup (though the GPU sometimes attains slowdown). Over three orders of magnitude is achieved by the FPGA implementation on average; here, speedup is minimal and over one order of magnitude for simple queries, followed by a steep speedup increase as the



(a) Trucks



(b) Geolife



(c) Buses

Fig. 16 FlexTrack (software) *IJP* throughput (MB/s) resulting from matching for 2,048 queries with varying properties on the **a** *Trucks*, **b** *GeoLife* and **c** *Buses* datasets

query complexity (number of variables, wildcard usage) increases. The GPU implementation results in to two orders magnitude speedup (Buses dataset), while slowdown is also shown (especially for less expressive queries).

7 Conclusions

The wide availability of applications that combine cellular and GPS technologies has created large trajectory depositories. Complex pattern queries provide an intuitive way to access relevant data as they can select trajectories based on specific events of interest. However, as the complexity of the posed pattern queries increases, so do computational requirements, which are not easily met using traditional CPU-based software platforms.

In this paper, we present the first proof-of-concept study on FPGA- and GPU-based solutions for parallel matching of variable-enhanced complex patterns, with a focus on stream-mode (single pass) filtering. We describe a tool for automatically generating hardware constructs using a set of pattern queries, abstracting away ramifications of hardware code development and deployment. We also present a GPU implementation of the same matching algorithms, used for hardware solution. A thorough design space exploration of the parallel architectures shows that the proposed approach offers

good scalability, fitting thousands of pattern query matching engines both on a Xilinx V6LX240T FPGA and on Nvidia K20 GPU. For FPGAs, increasing the number of variables and wildcards is shown to have linear effect on the resulting circuit size and negligible effect on performance. The same effect could be seen for the throughput of GPU-based implementation, prior to a "breaking point". However this behavior does not happen in CPU-based solutions, since performance is greatly affected from such pattern query characteristics. We show an extensive performance evaluation where FPGA and GPU setups outperform the current state-of-the-art (single-threaded) CPU-based approaches, by over three orders of magnitude for FPGAs (for expressive queries) and up to two orders of magnitude for certain datasets on GPUs (and in some cases slowdown).

When handling pattern queries with (a) no variables, (b) one variable, or (c) no wildcards with two or more variables, both proposed parallel implementations are able to process the trajectory data in a single pass. When two or more variables occur in a pattern query alongside wildcards, the proposed solution may have the drawback of resulting in false positive matches (though these are minimal in practice). Nonetheless, a no-false-positive solution is proposed, though being limited in scalability.

As part of our future research, we are working on enhancing the proposed FPGA-based solution to allow online pattern query updates. In this way, the deployed generic pattern query engines will support *any* pattern query structure and node values. A stream of bits forwarded to the FPGA will program the connections between deployed pattern query nodes. It should be noticed that this approach is different to the Dynamic Partial Reconfiguration (DPR), where the bit configuration of the FPGA itself is updated. For the GPU-based approach we are planning to leverage intra-trajectory parallelism, by executing different GPU matching kernels concurrently on the same graphical card.

Acknowledgments This work has been partially supported by National Science Foundation awards: CCF-1219180, IIS-1161997 and IIS-1305253.

References

1. Absalyamov I, Moussalli R, Tsotras VJ, Najjar WA (2013) High-performance XML twig filtering using GPUs. In: Proceedings of the ADMS, pp 13–24
2. Aggarwal C, Agrawal D (2003) On nearest neighbor indexing of nonlinear trajectories. In: Proceedings of the ACM PODS, pp 252–259
3. Bakkum P, Skadron K (2010) Accelerating SQL database operations on a GPU with CUDA. In: Proceedings of the GPGPU, pp 94–103. ACM
4. Beier F, Kilias T, Sattler KU (2012) GiST scan acceleration using coprocessors. In: Proceedings of the DaMoN, pp 63–69
5. Cascarano N, Rolando P, Risso F, Sisto R. (2010) iNFAnt: NFA pattern matching on GPGPU devices. SIGCOMM Comput Commun Rev 40(5):20–26
6. Cazalas J, Guha RK (2012) Performance Modeling of Spatio-Temporal Algorithms Over GEDS Framework. IJGHPC 4(3):63–84
7. (2013) Chorochronos: http://www.chorochronos.org
8. Erwig M, Schneider M (2002) Spatio-Temporal Predicates. IEEE Trans Knowl Data Eng 14(4):881–901
9. Fender J, Rose J (2003) A High-Speed Ray Tracing Engine Built on a Field-Programmable System. In: Proceedings of the IEEE FPT, pp 188–195
10. Hadjieleftheriou M, Kollios G, Bakalov P, Tsotras VJ (2005) Complex Spatio-temporal Pattern Queries. In: Proceedings of the VLDB, pp 877–888

11. Hadjieleftheriou M, Kollios G, Tsotras VJ, Gunopulos D (2006) Indexing Spatiotemporal Archives. VLDB J 15(2):143–164
12. He B, Yang K, Fang R, Lu M, Govindaraju N, Luo Q, Sander P (2008) Relational joins on graphics processors. In: Proceedings of the ACM SIGMOD, pp 511–524. ACM
13. Heckbert PS (1994) Graphics Gems IV, vol 4. Morgan Kaufmann
14. Kim C, et al. (2010) FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In: Proceedings of the ACM SIGMOD, pp 339–350. ACM
15. Kim SS, Nam SW, Lee IH (2007) Fast Ray-Triangle Intersection Computation Using Reconfigurable Hardware. In: Computer Vision/Computer Graphics Collaboration Techniques, LNCS, vol 4418, pp 70–81. Springer
16. Knuth D, Morris J, Pratt V (1977) Fast Pattern Matching in Strings. SIAM. J Comput 6(2):323–350
17. Kumar S et al (2006) Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In: Proceedings of the ACM SIGCOMM, pp 339–350
18. Mitra A, Najjar W, Bhuyan L (2007) Compiling PCRE to FPGA for Accelerating SNORT IDS. In: Proceedings of the ACM/IEEE ANCS, pp 127–136
19. Mokhtar H, Su J, Ibarra O (2002) On Moving Object Queries. In: Proceedings of the ACM PODS, pp 188–198
20. Moussalli R, Halstead R, Salloum M, Najjar W, Tsotras VJ (2011) Efficient XML path filtering using GPUs. In: Proceedings of the ADMS
21. Moussalli R, Najjar W, Luo X, Khan A (2013) A High Throughput No-Stall Golomb-Rice Hardware Decoder. In: Proceedings of the IEEE FCCM, pp 65–72
22. Moussalli R, Salloum M, Najjar W, Tsotras VJ (2010) Accelerating XML Query Matching Through Custom Stack Generation on FPGAs. In: HiPEAC, pp 141–155
23. Moussalli R, Salloum M, Najjar W, Tsotras VJ (2011) Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In: Proceedings of the IEEE ICDE
24. Moussalli R, Vieira MR, Najjar WA, Tsotras VJ (2013) Stream-mode fpga acceleration of complex pattern trajectory querying. In: Proceedings of the SSTD, pp 201–222
25. Mouza C, Rigaux P (2005) Mobility Patterns. Geoinformatica 9(4):297–319
26. du Mouza C, Rigaux P, Scholl M (2005) Efficient evaluation of parameterized pattern queries. In: Proceedings of the ACM CIKM, pp 728–735
27. Nvidia (2014) Nvidia GPU Programming Guide. http://docs.nvidia.com/cuda/
28. Pfoser D, Jensen C, Theodoridis Y (2000) Novel Approaches in Query Processing for Moving Object Trajectories. In: Proceedings of the VLDB, pp 395–406
29. Pico Computing M-Series Modules (2012). http://picocomputing.com/m-series/m-501
30. Piorkowski M, Sarafijanovoc-Djukic N, Grossglauser M (2009) A Parsimonious Model of Mobile Partitioned Networks with Clustering. In: Proceedings of the COMSNETS
31. Sadoghi M, et al. (2010) Efficient Event Processing Through Reconfigurable Hardware for Algorithmic Trading. Proc. VLDB Endow 3(1–2):1525–1528
32. Sakr MA, Güting RH (2009) Spatiotemporal Pattern Queries in Secondo. In: Proceedings of the SSTD, pp 422–426
33. Schmittler J, Woop S, Wagner D, Paul WJ, Slusallek P (2004) Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In: Proceedings of the ACM HWWS, pp 95–106
34. Sidhu R, Prasanna VK (2001) Fast Regular Expression Matching Using FPGAs. In: Proceedings of the IEEE FCCM, pp 227–238
35. Tao Y, Papadias D (2001) MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In: Proceedings of the VLDB, pp 431–440
36. Tao Y, Papadias D, Shen Q (2002) Continuous Nearest Neighbor Search. In: Proceedings of the VLDB, pp 287–298
37. Teubner J, Müller R, Alonso G (2010) FPGA Acceleration for the Frequent Item Problem. In: Proceedings of the IEEE ICDE, pp 669–680
38. Vieira MR, Bakalov P, Tsotras VJ (2010) Querying Trajectories Using Flexible Patterns. In: Proceedings of the EDBT, pp 406–417
39. Vieira MR, Bakalov P, Tsotras VJ (2011) FlexTrack: a System for Querying Flexible Patterns in Trajectory Databases. In: Proceedings of the SSTD, pp 475–480
40. Woods L, Teubner J, Alonso G (2010) Complex Event Detection at Wire Speed with FPGAs. Proc. the VLDB Endow 3(1–2):660–669
41. Zheng Y, Xie X, Ma WY (2010) GeoLife: A Collaborative Social Networking Service Among User, Location and Trajectory. IEEE Data Eng Bull 33(2):32–40

Roger Moussalli is a Research Staff Member with the Digital Communications IC design group at the IBM T.J. Watson Research Center, since April 2013. His research interests include FPGA-based acceleration, computer architecture and big data analytics.

Prior to joining IBM Research, Roger received his Ph.D. in Computer Science (2013) from the University of California, Riverside, and B.E. in Electrical and Computer Engineering (2007) from the American University of Beirut, Lebanon.

Ildar Absalyamov is a second year PhD student in the Department of Computer Science & Engineering at the University of California, Riverside. His research interests are in the area of databases, big data analytics and distributed systems.

Prior to coming to UCR he has received CS degree in Computer Science from the Ufa State Aviation Technical University, Russia (2011).

Marcos R. Vieira has been a Research Staff Member with the Natural Resources Analytics Group at IBM Research - Brazil, since April 2012. His research interests include spatio-temporal databases, indexing and query optimization techniques, and big data analytics.

Prior to joining IBM Research, Marcos received his Ph.D. in Computer Science (2011) from the University of California, Riverside, M.Sc. in Computer Science (2004) from the University of Sao Paulo at Sao Carlos, Brazil, and B.E. in Computer Engineering (2001) from Federal University of Sao Carlos, Brazil.

Walid A. Najjar is a Professor in the Department of Computer Science and Engineering at the University of California Riverside. His areas of research include computer architectures and compilers for parallel and high-performance computing, embedded systems, FPGA-based code acceleration and reconfigurable computing.

Walid received a B.E. in Electrical Engineering from the American University of Beirut in 1979, and the M.S. and Ph.D. in Computer Engineering from the University of Southern California in 1985 and 1988 respectively. From 1989 to 2000 he was on the faculty of the Department of Computer Science at Colorado State University, before that he was with the USC-Information Sciences Institute. He was elected Fellow of the IEEE and the AAAS.

Vassilis J. Tsotras is a Professor at the Department of Computer Science, University of California, Riverside. He received his Diploma from the National Technical University of Athens and his Ph.D. from Columbia University. His research interests include temporal and spatiotemporal databases, semistructured data management and data dissemination. He received the NSF Research Initiation Award (1991) and the Teaching Excellence Award from the Bourns College of Engineering (1999). He has served as the General Co-Chair for the 26th International Conference on Data Engineering (ICDE'10), General Chair for the 7th Symposium on Spatial and Temporal Databases (SSTD'01) and Program Chair (Database Track) for the ACM 15th Conference on Information and Knowledge Management (CIKM'06). He is co-Editor in Chief of the International Journal of Cooperative Information Systems and served on the editorial board of the Very Large Databases Journal and IEEE Transactions on Knowledge and Data Engineering (TKDE). Prof. Tsotras research has been supported by various grants from NSF, the Department of Defense and the industry.