Stream-Mode FPGA Acceleration of Complex Pattern Trajectory Querying

Roger Moussalli[1], Marcos R. Vieira[2], Walid Najjar[1], and Vassilis J. Tsotras[1]

[1] University of California, Riverside, USA
{rmous,najjar,tsotras}@cs.ucr.edu
[2] IBM Research, Brazil
mvieira@br.ibm.com

Abstract. The wide and increasing availability of collected data in the form of trajectory has lead to research advances in behavioral aspects of the monitored subjects (e.g., wild animals, people, 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-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 setups outperform the current state-of-the-art CPU-based approaches by over three orders of magnitude. Unlike software-based approaches, the performance of the proposed FPGA solution is only minimally affected by the increased pattern complexity.

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 execute 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.

M.A. Nascimento et al. (Eds.): SSTD 2013, LNCS 8098, pp. 201–222, 2013.

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 advantages on performance of such platforms arise from the 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.

As traditional platforms are increasingly hitting limitations when processing large volumes of streaming data, researchers are investigating FPGAs for database applications. Recent work has focused on the adoption of FPGAs for data stream processing in different scenarios. In [18] a stream filtering approach is presented for XML documents. [30] investigated the speedup of the frequent item problem using FPGAs. In [33], the FPGA is employed for complex event detection using regular expressions. [23] proposed a predicate-based filtering on FPGAs where user profiles are expressed as a conjunctive set of boolean filters. [16] describes an FPGA-based stream-mode decompression engine targeting Golomb-Rice encoded inverted indexes.

In this paper, we describe an FPGA-based setup allowing users to query spatio-temporal databases in a very powerful and intuitive way. Figure 1 depicts a generic overview of the various steps performed in spatio-temporal querying setups. 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 [31,32], 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

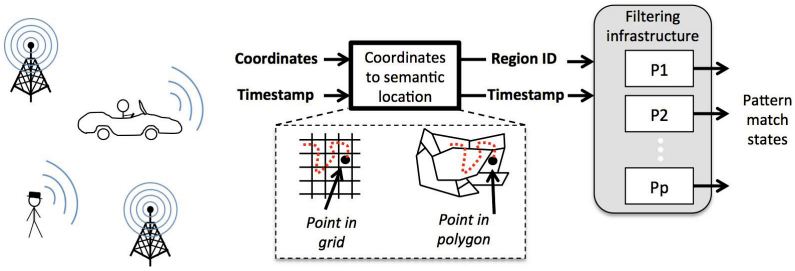


Fig. 1. Generic overview of various steps performed in spatio-temporal querying setups

entire life span of the trajectory (e.g., "*Find all trajectories that are similar to a given query trajectory according to some similarity measure*").

Flexible pattern queries can also have "variable" spatial predicates, and thus substantially enhancing the flexibility and expressive power of the *FlexTrack* framework. An example of a variable-enhanced query is "*Find all trajectories that started in a region, then visited the downtown Munich, then at some later point returned to the first visited region*".

This work serves as a proof-of-concept on the performance benefits of evaluating flexible pattern queries using FPGAs. Here we focus on the challenges of supporting hundreds (up to thousands) of variable-enhanced flexible patterns on FPGAs in streaming (fully-pipelined) fashion. Using FPGAs 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. 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 support for variable-enhanced 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 querying architecture is detailed in Section 4; the experimental evaluation is provided in Section 5; and the conclusions appear in Section 6.

2 Related Work

Single predicate queries (e.g., Range and *NN* queries) for trajectory data have been widely studied in the past (e.g., [2,20,28]). In order to make the query evaluation process more efficient [8], trajectories are first approximated using Minimum Bounding Regions (MBR) and then indexed using hierarchical spatiotemporal indexing structures, like the MVR-tree [27] and TPR-tree [29]. 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 [5,14,24], as well as query evaluation algorithms [7,4,19].

The *FlexTrack* system [31,32], which our work is based on, provides a more general and powerful query framework than previous approaches. 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 recently explored by many studies [26,13,12,33]. Most of these works focus on deep packet inspection and security as applications of interest. Using FPGAs, speedups of up to two orders of magnitude is achieved compared to CPU-based approaches, as every data element in stream can be processed in a single hardware cycle. The works in [17,15,18] 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 bases approaches.

In [26] an NFA implementation of regular expressions on FPGAs is described. [13] proposes generating hardware code from Perl Compatible Regular Expressions. The work in [12] focuses on DFA implementations of regular expressions, while merging commonalities among multiple DFAs. [33] proposes the use of regular expressions for the representation of spatio-temporal queries. An FPGA implementation is detailed, allowing the sharing of query evaluation engines among several trajectories, with a minor impact on performance. In [3], it is investigated the use of GPUs for the fast computation of proximity area views over streams of spatio-temporal data. 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-based support of variable-enhanced patterns.

3 The *FlexTrack* System

We now provide a briefly description of the pattern query language syntax, as well as the key elements in the *FlexTrack* framework (for more details, see [31,32]).

3.1 Flexible Pattern Query Language

A trajectory T_{id} is defined as a list of locations collected for a specific moving object over an ordered sequence of timestamps, and is stored as a sequence of n pairs $\{(ls_1, ts_1), \ldots (ls_n, ts_n)\}$, where $ls_i \in \mathbb{R}^d$ is the object location recorded at timestamp ts_i $(ts_{i-1} < ts_i)$.

The *FlexTrack* uses a set of non-overlapping regions Σ_l that are derived from partitioning the spatial domain. 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 interval $t := (t_{from} : t_{to}) \mid t_s \mid t_r$. 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, constrains 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 $\mathcal{S}$. 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 $\mathcal{I}$, 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*.

In order to fast prune trajectories that do not satisfy pattern $\mathcal{S}$ the *T-list* is used. 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 [11] 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 Hardware Solution

4.1 Compiling Queries to Hardware

In this work, 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 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

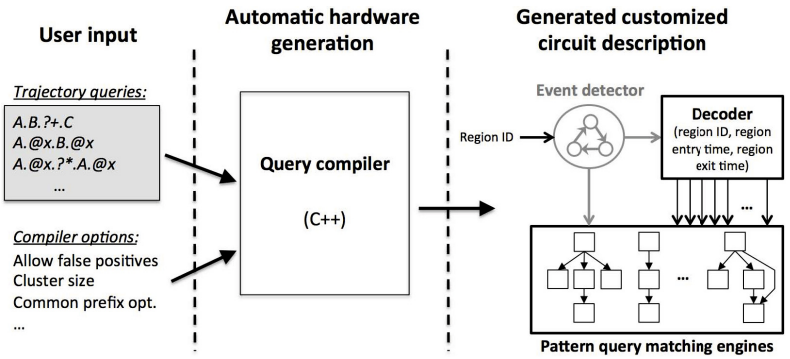


Fig. 2. Query-to-hardware tool flow

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.

4.2 High Level Architecture Overview

As depicted in Fig. 2, assuming an input stream of pairs $\langle location,\ timestamp \rangle$, 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 are 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 [6,9,10,25]). 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, 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.

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.

4.3 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 [13,26]. Figure 3(a) depicts the matching engine respective to the pattern query $A.B.?^*.A$, and Fig. 3(b) 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, depicted in grey in Fig. 3(b)), 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:

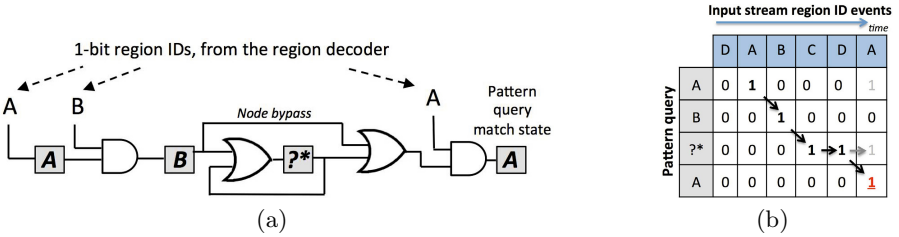


Fig. 3. (a) Query matching engines respective to the pattern query $A.B.?^*.A$, and (b) an event-by-event overview of the matching of the query.

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. 3(a));
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 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. 3(a)). 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. 3(a)).

Looking closer at Fig. 3(b), 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. 3(b) indicates that the query was detected in the input stream.

4.4 Evaluating Patterns with Variables and without Wildstar/Wildplus Predicates

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.

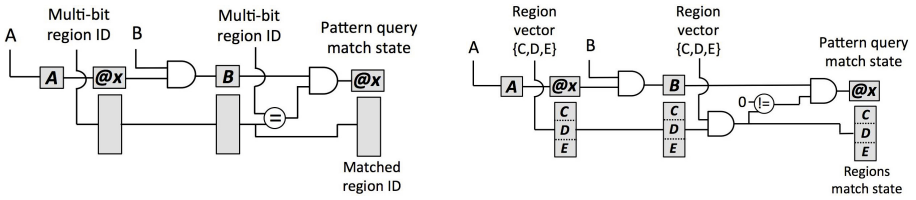


Fig. 4. Query matching engines respective to the pattern query $A.@x.B.@x$, (a) without and (b) with a region set constraint $\{C, D, E\}$ on $@x$. To handle variables in hardware, the first instance of a given variable in a query forwards, alongside the incoming match state, (a) the event detector's output encoded (multi-bit) region-ID, and (b) a one-hot signal consisting of bits respective to each region in the set of the variable. Every later instance of that variable in the query (here, the last query node) would match the event detector's ((a) encoded, and (b) multiple decoded) 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.

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 4.5 and 4.6. 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 match state (see the second node in Fig. 4(a)). 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.

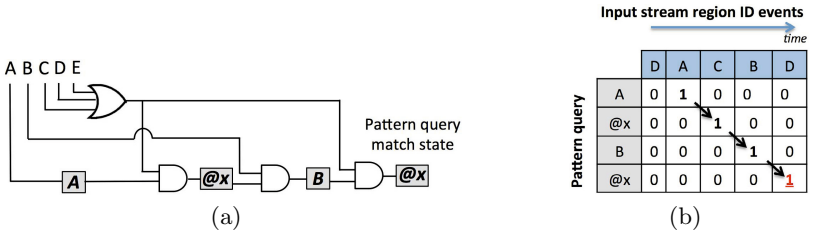


Fig. 5. (a) Query matching engine respective to the pattern query $A.@x.B.@x$ $\{@x : C, D, E\}$, such that the variable region set constraint is implemented as a "relaxed" OR. This relaxation helps save considerable hardware resources (compare to Fig. 4(b)). (b) An event-by-event overview of the matching of the query resulting in a false positive, due to the OR-based implementation of the variable region set constraint.

The same solution is applicable to pattern queries containing variables with region sets. Figure 4(b) 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. 4(b) 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 5(a) 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. 5(b)), the 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.

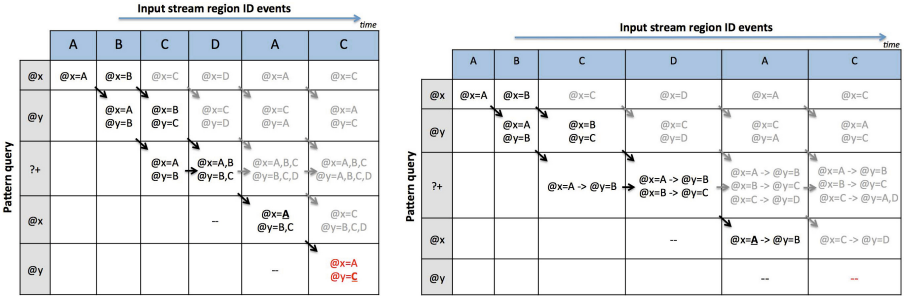


Fig. 6. Event-by-event matching of the pattern query $@x.@y.?^{+}.@x.@y$ $\{@x : A, B, C, D\}$ $\{@y : A, B, C, D\}$. The resulting match in (a) is a false positive; whereas enough state is saved in (b) at the aggregator node ($?^{+}$) to eliminate that false positive.

4.5 Evaluating Patterns with a Single Variable and with Wildstar/Wildplus Predicates

The remainder of this discussion is applicable to both wildplus and wildstar query nodes. As detailed earlier (Fig. 3(a)), 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.

4.6 Evaluating Patterns with Multiple Variables and with Wildstar/Wildplus Predicates

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 positve matches, as even more state should be saved than discussed earlier. Figure 6(a) 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 Figure 6(a), notice that the variable buffers for both $@x$ and $@y$ hold an increasing 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. 6(b)). 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. 6(a).

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:

1. $\mathcal{S}_1 = A.C.?^+.A.C$
2. $\mathcal{S}_2 = A.D.?^+.A.D$
3. $\mathcal{S}_3 = B.C.?^+.B.C$
4. $\mathcal{S}_4 = B.D.?^+.B.D$

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 approach, a study on the resulting false positives versus resource utilization is performed in Section 5. 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 Experimental Evaluation

We now present an extensive experimental evaluation of the proposed hardware architecture. 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 a study on matching accuracy. Finally, we show the performance evaluation between the proposed architecture solutions with the CPU-based software approach.

5.1 Dataset Description

In our experimental evaluation, we use four real trajectory datasets. The first two datasets are the *Trucks* and *Buses* from [1]. 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 [22] 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 [34] during a period of over three years. This dataset has 17,621 trajectories with a total distance of about 1.2 million kilometers and duration of more than 48,000 hours.

5.2 Experiments 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.

In order 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 each variable and wildcard inside the query are 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 [21]. 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 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.

5.3 Design Space Exploration

Here we discuss the resource utilization and achievable performance (throughput) of the hardware engines. Figure 7(a) shows the resource utilization, and Fig. 7(b) shows the respective frequencies of the hardware engines, such that the number of queries (varying from 32, 64, 128, ... up to 2,048 queries), the query length (4 and 8 predicates), and number of variables in a pattern query (0 and 1 variable, in this last case a variable with a region set of 5 regions is assumed).

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. While these numbers address the scalability of the proposed matching engines, Fig. 7(b) details the respective achievable performance in terms of:

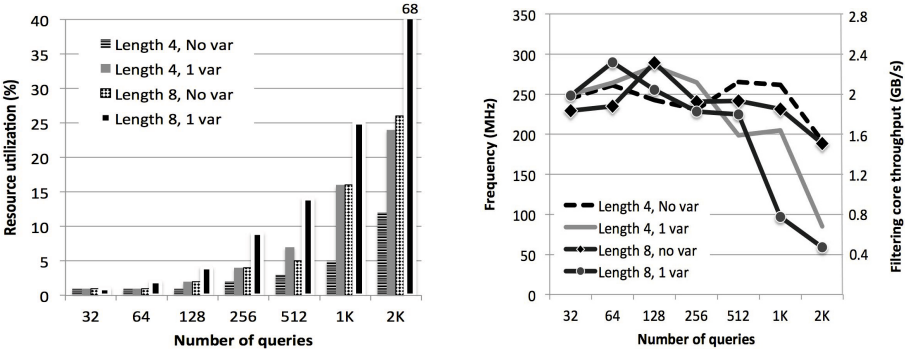


Fig. 7. (a) Resource utilization and (b) respective frequencies/throughput of the hardware engines, such that the number of queries is doubled, the query length is doubled, and variable predicate is present or not in the pattern query

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

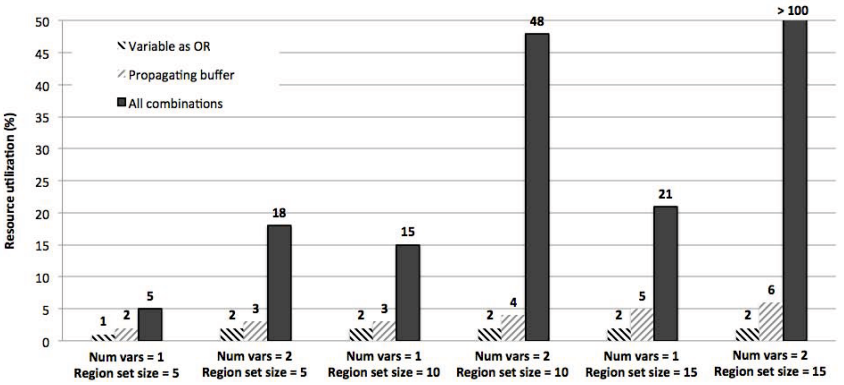


Fig. 8. Scalability of the each of the following three implementations of 100 queries of length 6 holding variables: **variable as OR**, **propagating buffer**, and **all combinations**

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).

5.4 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);
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 8 illustrates the resource utilization of 100 queries of length 6 holding variables for 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. 8), 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

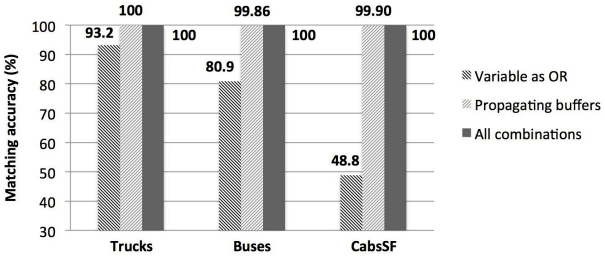


Fig. 9. Matching accuracy (100-false positives %) for each implementation of 100 long queries, over three datasets, namely *Trucks*, *Buses* and *CabsSF*

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.

We now evaluate the number of false positive matches for each of the three query engine implementations previously discussed. In this experiment, as shown in Fig. 9, the matching accuracy (100-false positives %) 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). 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 by its design, 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 accuracy varies from high (93.2%), to 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*.

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

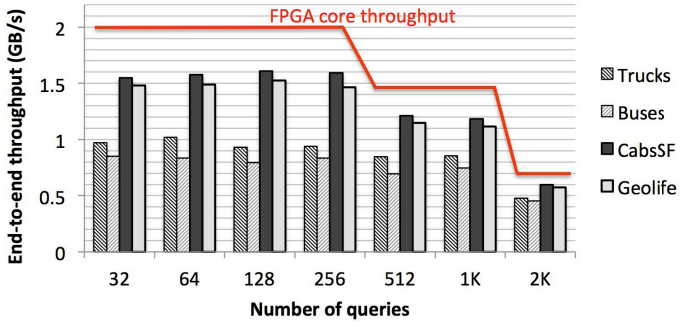
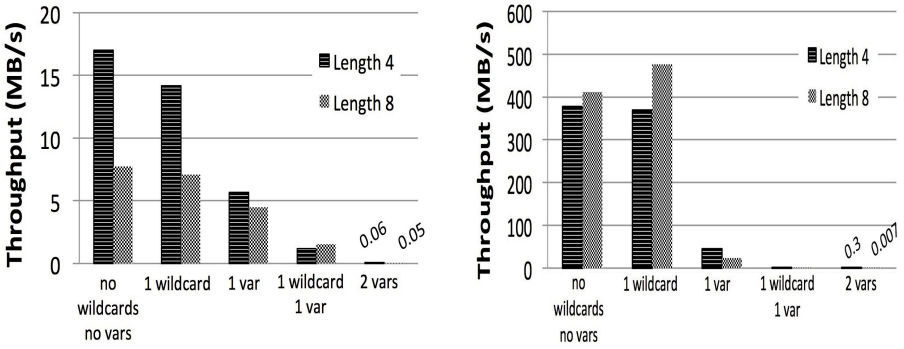


Fig. 10. End-to-end (CPU-RAM to FPGA and back) throughput of queries of length 4 with 1 variable. The throughput of the FPGA filtering core is drawn in red line.



(a) Trucks (b) Geolife

Fig. 11. FlexTrack (software) *IJP* throughput (MB/s) resulting from matching for 2,048 queries with varying properties on the (a) *Trucks* and (b) *GeoLife* datasets. Increasing query complexity (adding variables/wildcards) greatly decreases throughput.

5.5 Performance Evaluation

In the last set of experiments, we compare the performance evaluation between our proposed architecture solutions and the CPU-based software approach. Figure 10 shows the end-to-end (CPU-RAM to FPGA and back) 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 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 11 depicts the FlexTrack (software) *IJP* throughput (MB/s) resulting from matching for 2,048 queries with varying properties on the Fig. 11(a) *Trucks* and Fig. 11(b) *GeoLife* datasets. Pre-processing (index building) time is excluded. When considering simple queries, throughput is initially higher for

the larger dataset (*GeoLife*), where processing steady-state is reached. Increasing query complexity (adding variables/wildcards) greatly decreases throughput. Note that where the FPGA end-to-end execution time is in the milliseconds range, software operates in the tens of seconds (up to several minutes) range, and is greatly affected by the query structure and dataset contents; hence the considerable speedup (over three orders of magnitude) and benefits of the FPGA setup. It should be noted that the proposed FPGA solution does not result in false positive matches for any of the queries considered in Fig. 11.

6 Conclusions

The wide and increasing availability of collected data in the form of trajectory has lead to research advances in behavioral aspects of the monitored subjects. 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. 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-based architectures for matching 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. A thorough design space exploration of the hardware architectures shows that the proposed solution offers good scalability, fitting thousands of pattern query matching engines on a Xilinx V6LX240T FPGA, a mid- to low-size FPGA. Increasing the number of variables and wildcards is shown to have linear effect on the resulting circuit size, and negligible on performance. This behavior does not happen in CPU-based solutions, since performance is greatly affected from such pattern query characteristics.

When handling pattern queries with (a) no variables, (b) one variable, or (c) no wildcards with two or more variables, the proposed hardware architecture is 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 framework 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.

Acknowledgments. This work was partially supported by NSF grants IIS-1144158 and IIS-1161997.

References

1. Chorochronos (2013), `http://www.chorochronos.org`
2. Aggarwal, C., Agrawal, D.: On nearest neighbor indexing of nonlinear trajectories. In: Proc. ACM Symp. on Principles of Database Systems (PODS), pp. 252–259 (2003)
3. Cazalas, J., Guha, R.: GEDS: GPU Execution of Continuous Queries on Spatio-Temporal Data Streams. In: IEEE/IFIP Int'l Conf. on Embedded and Ubiquitous Computing (EUC), pp. 112–119 (2010)
4. du Mouza, C., Rigaux, P., Scholl, M.: Efficient evaluation of parameterized pattern queries. In: Proc. ACM Int'l Conf. on Information and Knowledge Management (CIKM), pp. 728–735 (2005)
5. Erwig, M., Schneider, M.: Spatio-Temporal Predicates. IEEE Trans. Knowl. Data Eng. 14(4), 881–901 (2002)
6. Fender, J., Rose, J.: A High-Speed Ray Tracing Engine Built on a Field-Programmable System. In: Proc. IEEE Int'l Conf. on Field-Programmable Technology (FPT), pp. 188–195 (2003)
7. Hadjieleftheriou, M., Kollios, G., Bakalov, P., Tsotras, V.J.: Complex Spatio-temporal Pattern Queries. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 877–888 (2005)
8. Hadjieleftheriou, M., Kollios, G., Tsotras, V.J., Gunopulos, D.: Indexing Spatiotemporal Archives. VLDB J. 15(2), 143–164 (2006)
9. Heckbert, P.S.: Graphics Gems IV, vol. 4. Morgan Kaufmann (1994)
10. Kim, S.-S., Nam, S.-W., Lee, I.-H.: Fast Ray-Triangle Intersection Computation Using Reconfigurable Hardware. In: Computer Vision/Computer Graphics Collaboration Techniques, pp. 70–81 (2007)
11. Knuth, D., Morris, J., Pratt, V.: Fast Pattern Matching in Strings. SIAM J. Comput. 6(2), 323–350 (1977)
12. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In: ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 339–350 (2006)
13. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for Accelerating SNORT IDS. In: ACM/IEEE Symp. on Architecture for Networking and Communications Systems (ANCS), pp. 127–136 (2007)
14. Mokhtar, H., Su, J., Ibarra, O.: On Moving Object Queries. In: Proc. ACM Symp. on Principles of Database Systems (PODS), pp. 188–198 (2002)
15. Moussalli, R., Halstead, R., Salloum, M., Najjar, W., Tsotras, V.J.: Efficient XML Path Filtering Using GPUs. In: Workshop on Accelerating Data Management Systems, ADMS (2011)
16. Moussalli, R., Najjar, W., Luo, X., Khan, A.: A High Throughput No-Stall Golomb-Rice Hardware Decoder. In: IEEE Annual Int'l Symp. on Field-Programmable Custom Computing Machines, FCCM (2013)
17. Moussalli, R., Salloum, M., Najjar, W., Tsotras, V.: Accelerating XML Query Matching through Custom Stack Generation on FPGAs. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 141–155. Springer, Heidelberg (2010)

18. Moussalli, R., Salloum, M., Najjar, W., Tsotras, V.J.: Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In: Proc. IEEE Int'l Conf. on Data Engineering (ICDE) (2011)
19. Mouza, C., Rigaux, P.: Mobility Patterns. Geoinformatica 9(4), 297–319 (2005)
20. Pfoser, D., Jensen, C., Theodoridis, Y.: Novel Approaches in Query Processing for Moving Object Trajectories. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 395–406 (2000)
21. Pico Computing M-Series Modules (2012), http://picocomputing.com/m-series/m-501
22. Piorkowski, M., Sarafijanovoc-Djukic, N., Grossglauser, M.: A Parsimonious Model of Mobile Partitioned Networks with Clustering. In: Int'l Communication Systems and Networks and Workshops (2009)
23. Sadoghi, M., Labrecque, M., Singh, H., Shum, W., Jacobsen, H.-A.: Efficient Event Processing Through Reconfigurable Hardware for Algorithmic Trading. Proc. VLDB Endow. 3(1-2), 1525–1528 (2010)
24. Attia Sakr, M., Güting, R.H.: Spatiotemporal Pattern Queries in Secondo. In: Mamoulis, N., Seidl, T., Pedersen, T.B., Torp, K., Assent, I. (eds.) SSTD 2009. LNCS, vol. 5644, pp. 422–426. Springer, Heidelberg (2009)
25. Schmittler, J., Woop, S., Wagner, D., Paul, W.J., Slusallek, P.: Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In: Proc. ACM Conf. on Graphics Hardware (HWWS), pp. 95–106 (2004)
26. Sidhu, R., Prasanna, V.K.: Fast Regular Expression Matching Using FPGAs. In: Proc. the Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM), pp. 227–238 (2001)
27. Tao, Y., Papadias, D.: MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 431–440 (2001)
28. Tao, Y., Papadias, D., Shen, Q.: Continuous Nearest Neighbor Search. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 287–298 (2002)
29. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In: Proc. Intl. Conf. on Very Large Data Bases (VLDB), pp. 790–801 (2003)
30. Teubner, J., Müller, R., Alonso, G.: FPGA Acceleration for the Frequent Item Problem. In: Proc. IEEE Int'l Conf. on Data Engineering (ICDE), pp. 669–680 (2010)
31. Vieira, M.R., Bakalov, P., Tsotras, V.J.: Querying Trajectories Using Flexible Patterns. In: Proc. Int. Conf. on Extending Database Technology (EDBT), pp. 406–417 (2010)
32. Vieira, M.R., Bakalov, P., Tsotras, V.J.: FlexTrack: a System for Querying Flexible Patterns in Trajectory Databases. In: Proc. Int'l Symp. on Advances in Spatial and Temporal Databases (SSTD), pp. 475–480 (2011)
33. Woods, L., Teubner, J., Alonso, G.: Complex Event Detection at Wire Speed with FPGAs. Proc. VLDB Endow. 3(1-2), 660–669 (2010)
34. Zheng, Y., Xie, X., Ma, W.-Y.: GeoLife: A Collaborative Social Networking Service Among User, Location and Trajectory. IEEE Data Engineering Bulletin 33(2), 32–40 (2010)