High-Performance Holistic XML Twig Filtering Using GPUs

Ildar Absalyamov, Roger Moussalli,
Walid Najjar and Vassilis Tsotras

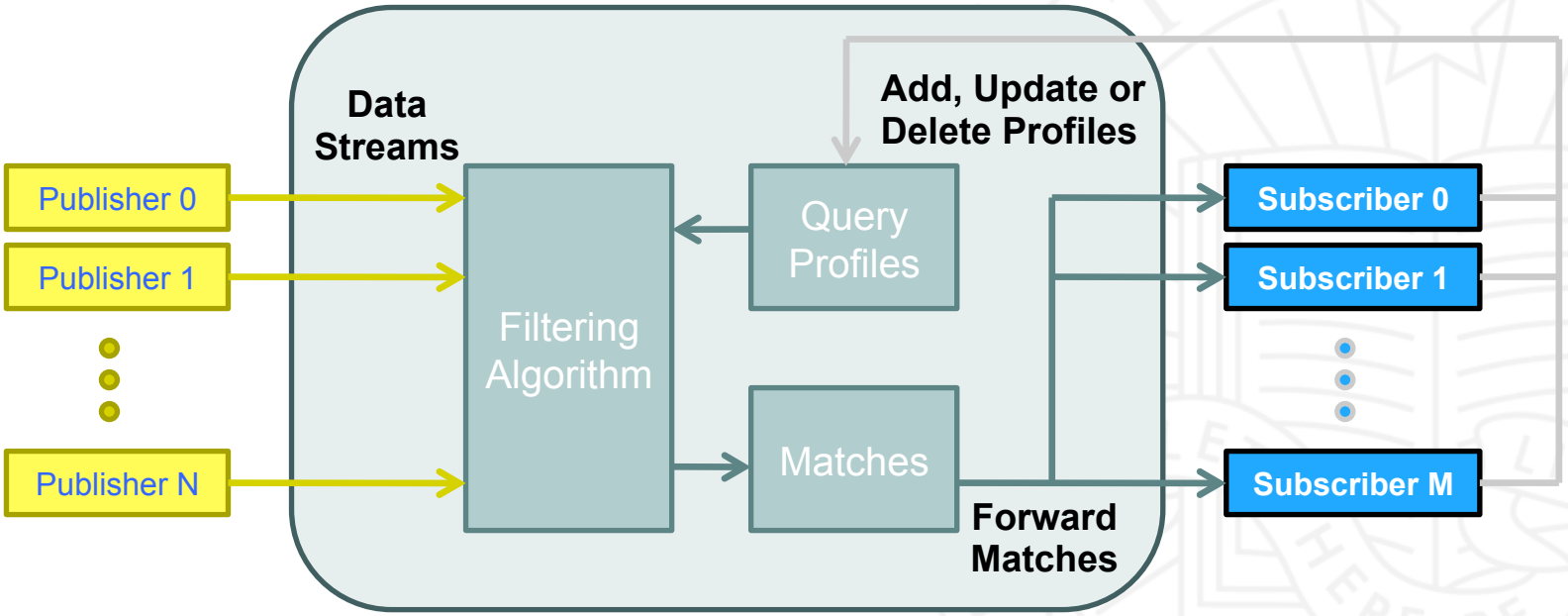
UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Outline

- Motivation
- XML filtering in the literature
  - Software approaches
  - Hardware approaches
- Proposed GPU-based approach & detailed algorithm
- Optimizations
- Experimental evaluation
- Conclusions

Motivation – XML Pub-subs

- Filtering engine is at the heart of publish-subscribe systems (pub-subs)
  - Used to deliver news, blog updates, stock data, etc
- XML is a standard format for data exchange
  - Powerful enough to capture message values as well as its structure using XPath
- The growing volume of information requires exploring massively parallel high-performance approaches for XML filtering



Related work (software)

- **XFilter** (VLDB 2000)
  - *Creates a separate FSM for each query*
- **YFilter** (TODS 2003)
  - *Combines individual paths, creates a single NFA*
- **LazyDFA** (TODS 2004)
  - *Uses deterministic FSMs*
- **XPush** (SIGMOD 2003)
  - *Lazily constructs a deterministic pushdown automaton*

FSM-based approaches

Related work (software)

- **FiST** (VLDB 2005)
  - *Converts the XML document into Prüfer sequences and matches respective sequences*
- **XTrie** (VLDB 2002)
  - *Uses Trie-based index to match query prefix*
- **AFilter** (VLDB 2006)
  - *Leverages prefix as well as suffix query indexes*

sequence-based approaches

others

Related work (hardware)

- **"Accelerating XML query matching through custom stack generation on FPGAs"** (HiPEAC 2010)
  - *Introduced dynamic-programming XML path filtering approach for FPGAs*
- **"Efficient XML path filtering using GPUs"** (ADMS 2011)
  - *Modified original approach to perform path filtering on GPUs*
- **"Massively parallel XML twig filtering using dynamic programming on FPGAs"** (ICDE 2011)
  - *Extended algorithm to support holistic twig filtering on FPGAs*

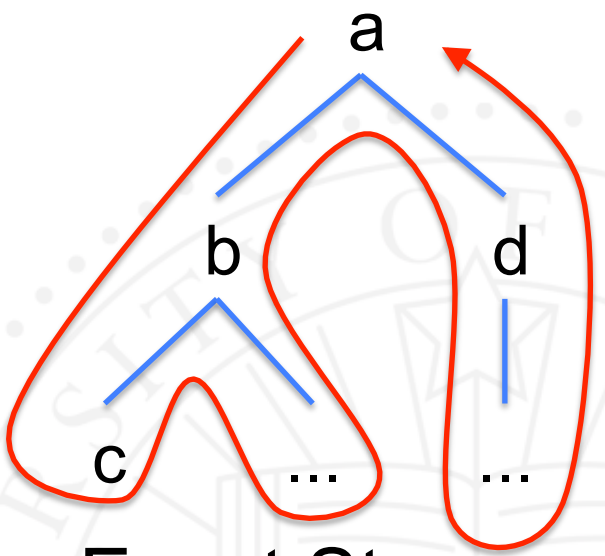
Why GPUs

- This work proposes holistic XML twig filtering algorithm, which runs on GPUs
- Why GPUs?
  - Highly **scalable**, massively parallel architecture
  - **Flexibility** as for software XML filtering engines
- Why not FPGAs?
  - Limited scalability due to scarce hardware resources available on the chip
  - Lack of query dynamicity - need time to reconfigure FGPA hardware implementation

XML Document Preprocessing

- To be able to run algorithm in streaming mode XML tree structure needs to be flattened
- XML document is presented as a stream of *open(tag)* and *close(tag)* events

XML Document

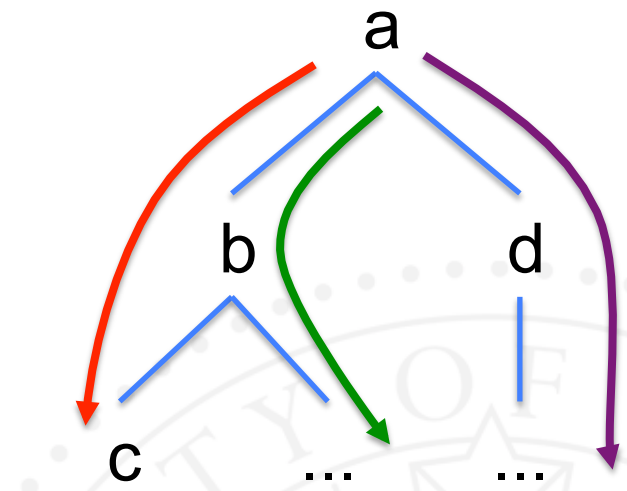


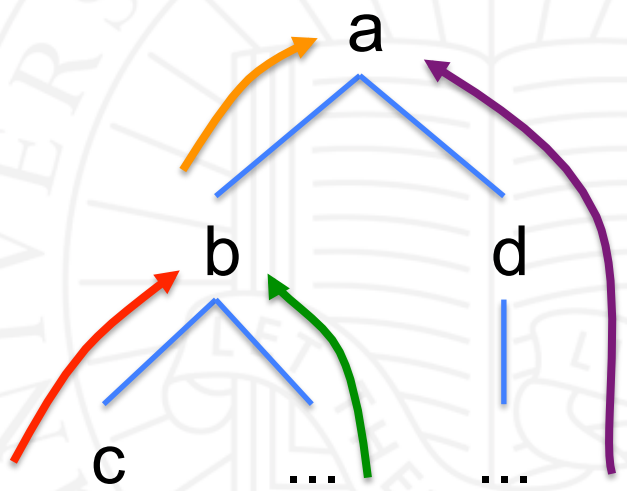
Event Stream

open(a) – open(b) – open(c) – close(c) – … – close(b) – open(d) – … – close(d) – close(a)

Twig Filtering: approach

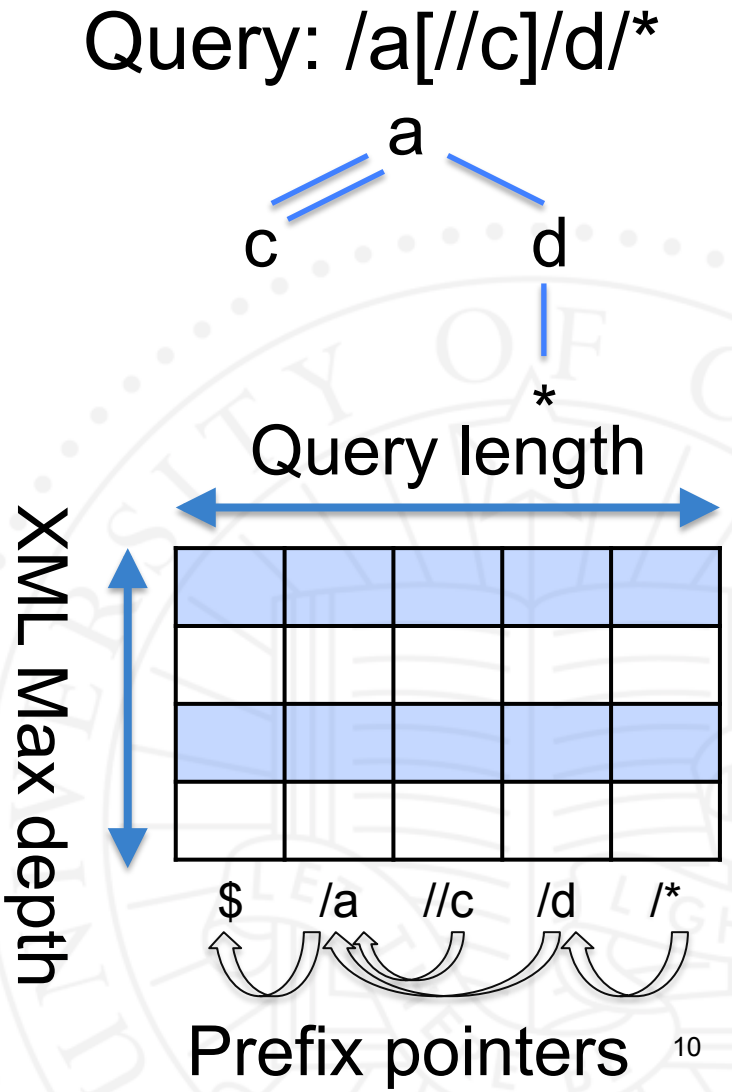
- Twig processing contains two steps
  - Matching individual root-to-leaf paths
  - Report matches back to root, while joining them at split nodes





Dynamic programming: algorithm

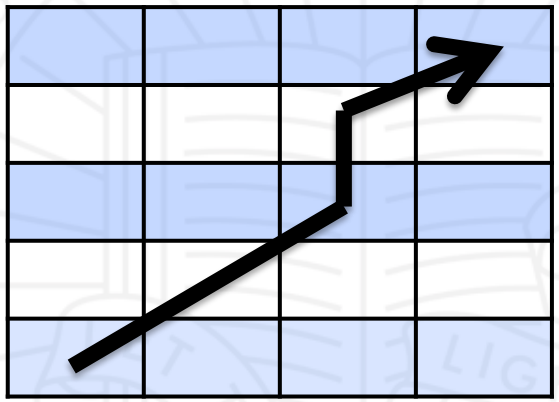
- Every query is mapped to a DP table
- DP table - binary stack
- Each stack is mapped to query
- Each node in query is mapped to stack column
- Every column has prefix pointer
- *Open* and *close* events map to push and pop actions on the top-of-the-stack (TOS)



Dynamic programming: stacks

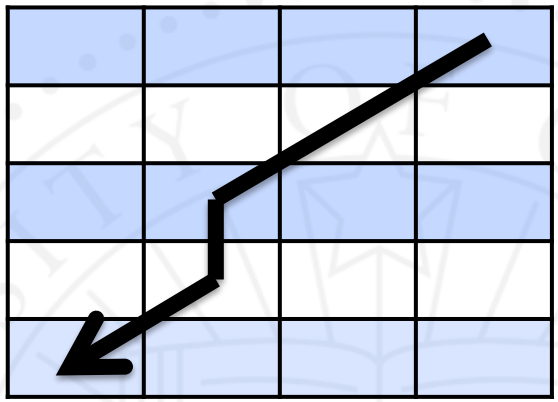
- Two different types of stack are used for different parts of filtering algorithm
- Stacks captures query matches via **propagation** of ‘1’s

- push stacks:
  - Used for matching root-to-leaf paths
  - The TOS is updated on open (i.e. push) events
  - Propagation goes diagonally upwards and vertically upwards from query root to query leaves



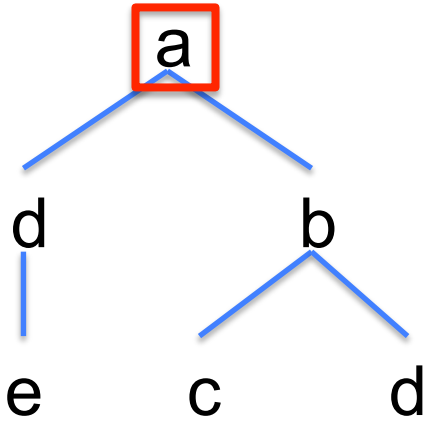
Dynamic programming: stacks

- pop stacks:
  - Used for reporting leaf matches back to the root
  - The TOS is updated on close (pop) events (open events clear the TOS)
  - Propagation goes diagonally downwards and vertically downwards from query leaves to the root

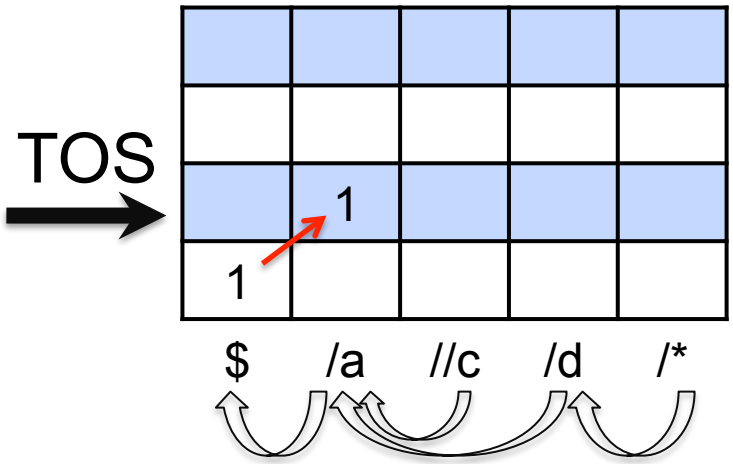


Push stack: Example

XML Document



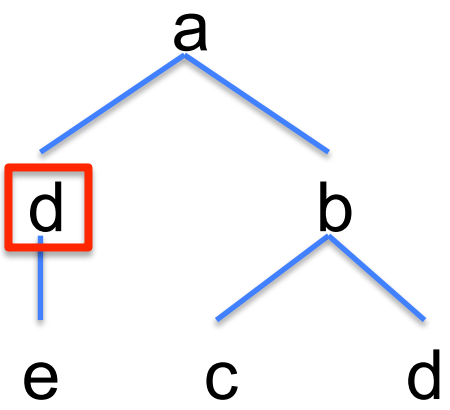
Open(a)



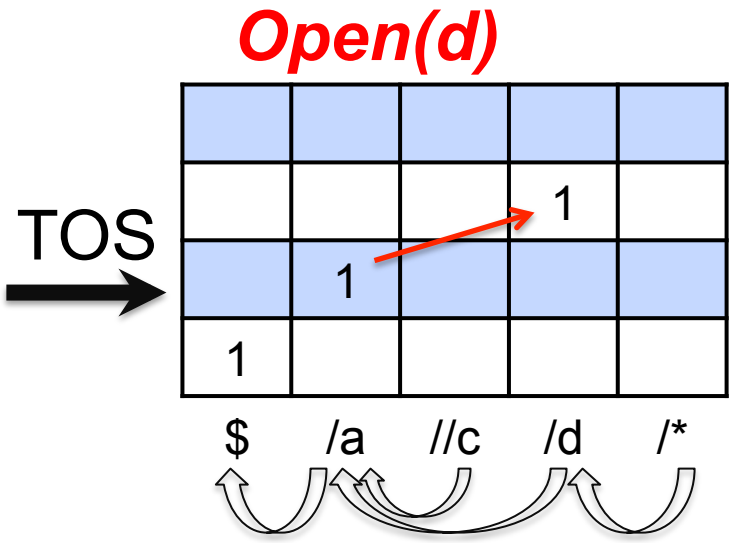
- Dummy root node ('$') is always matched in the beginning
- '1' is propagated diagonally upwards if
  - Prefix holds '1'
  - Relationship with prefix is '/'
  - *Open* event tag matches column tag

Push stack: Example

XML Document

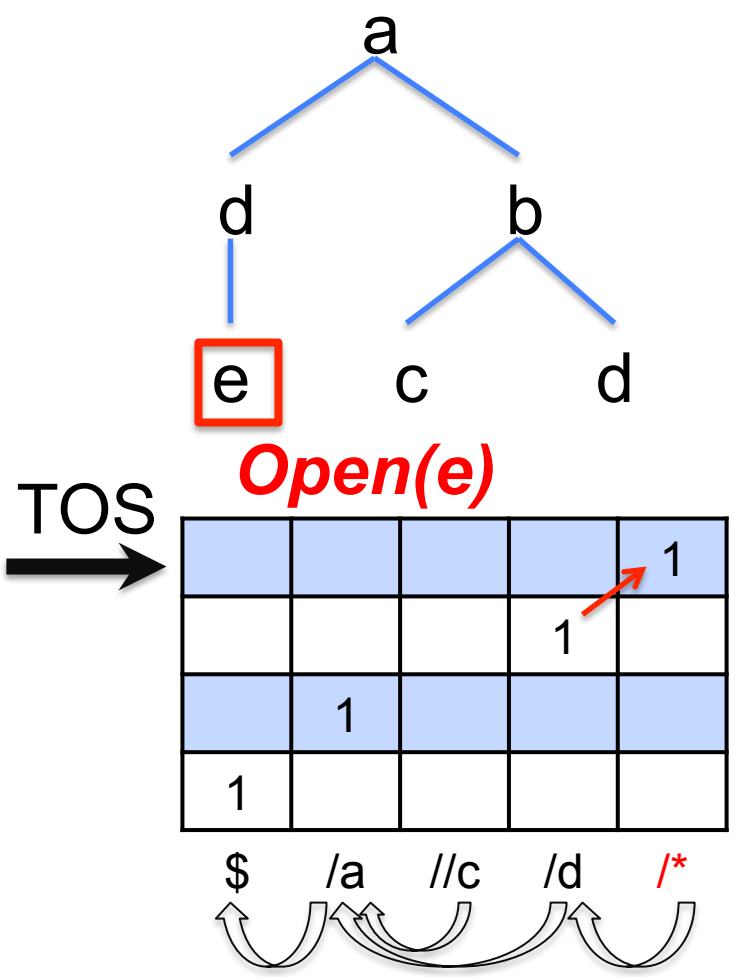


- '1' is propagated diagonally (in terms of prefix pointer) upwards as in previous example



Push stack: Example

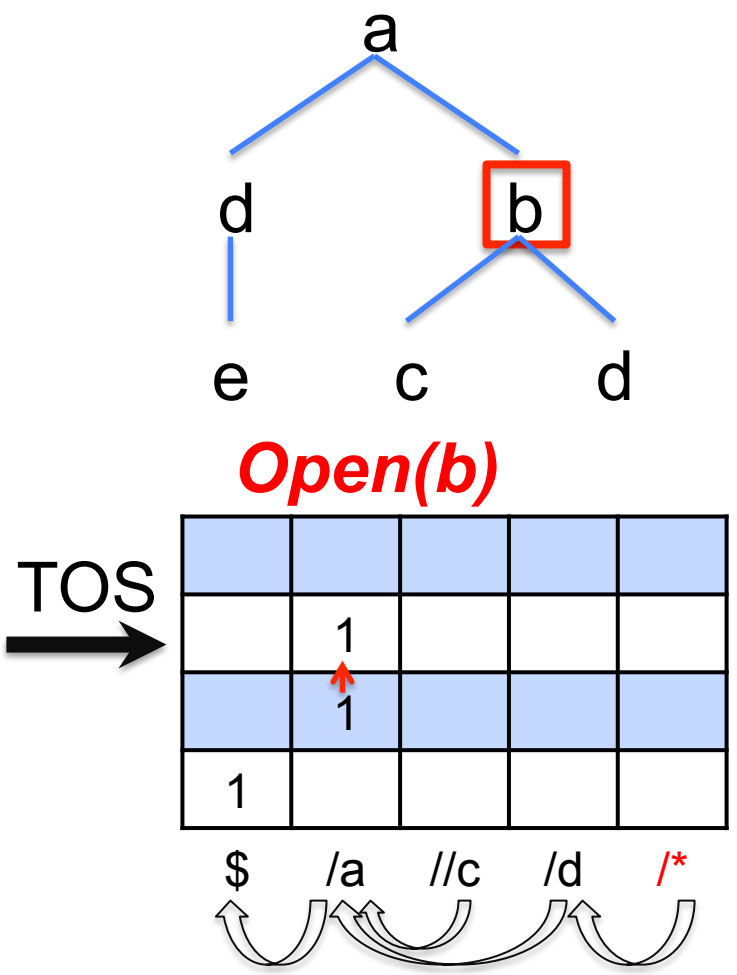
XML Document



- If the query node tag is a wildcard ('*'), then any tag in *open* event qualifies to be matched
- Since '/*' is a leaf node matched this fact is saved in a special binary array (colored in red in example)

Push stack: Example

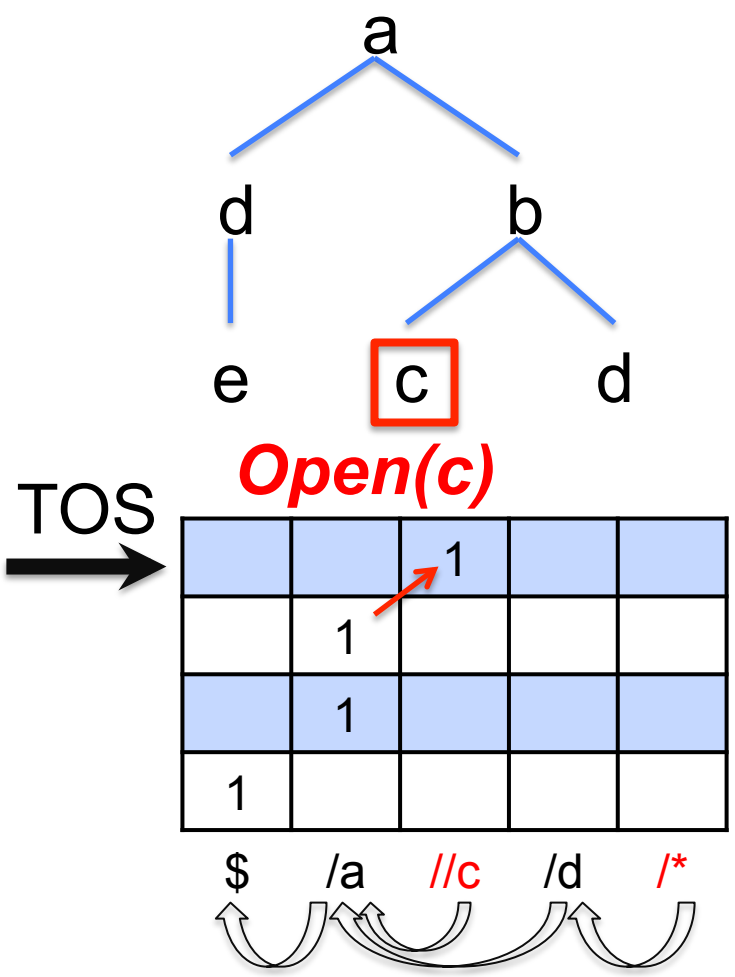
XML Document



- '1' propagates upwards in prefix column if
  - Prefix holds '1'
  - Relationship with prefix is '//'
  - Tag in *open* event could be arbitrary

Push stack: Example

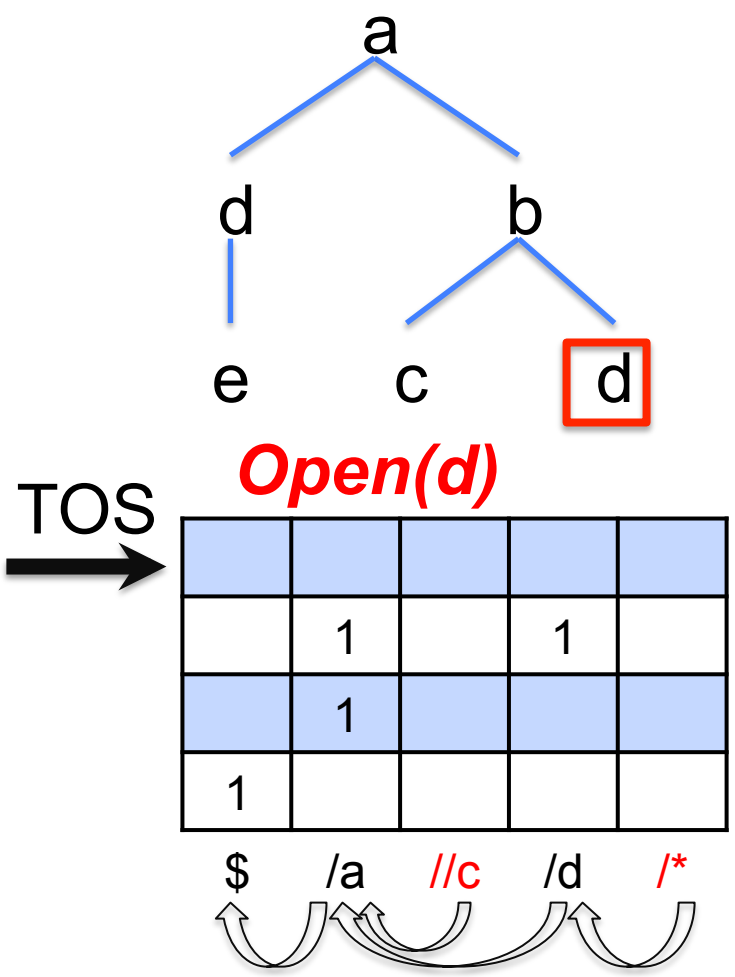
XML Document



- If '1' is propagated to query leaf node ('//c' in example) is saved as matched

Push stack: Example

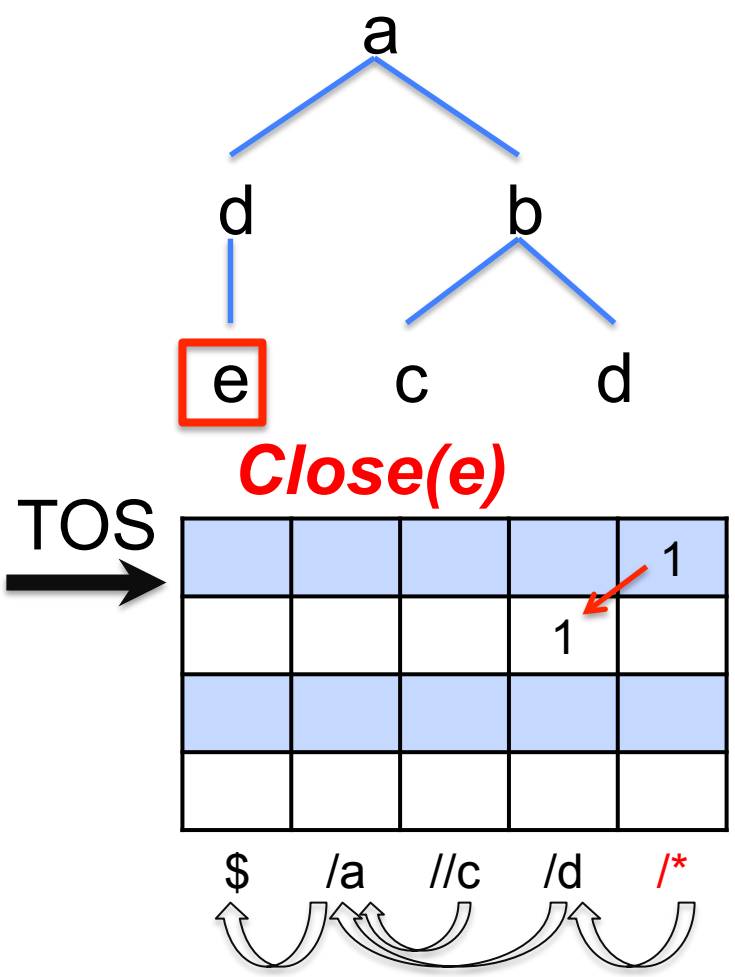
XML Document



- Node '/d' is not updated, since '/a' is a split node, whose children have different relationships ('//' with 'c' and '/' with 'd')
- The split node maintain different fields for these two kinds of children

Pop stack: Example

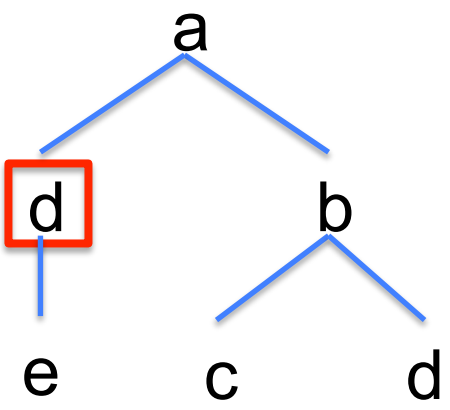
XML Document

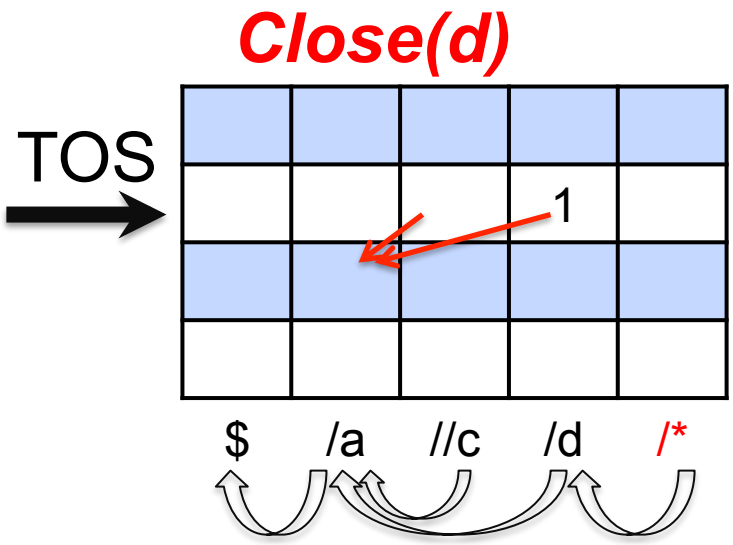


- Leaf nodes contain '1' if these nodes were saved in binary match array during 1st algorithm phase
- '1' is propagated diagonally downwards if
  - Node holds '1' on TOS
  - Relationship with prefix is '/'
  - *Close* event tag matches column tag or column tag is '*' (shown in example)

Pop stack: Example

XML Document

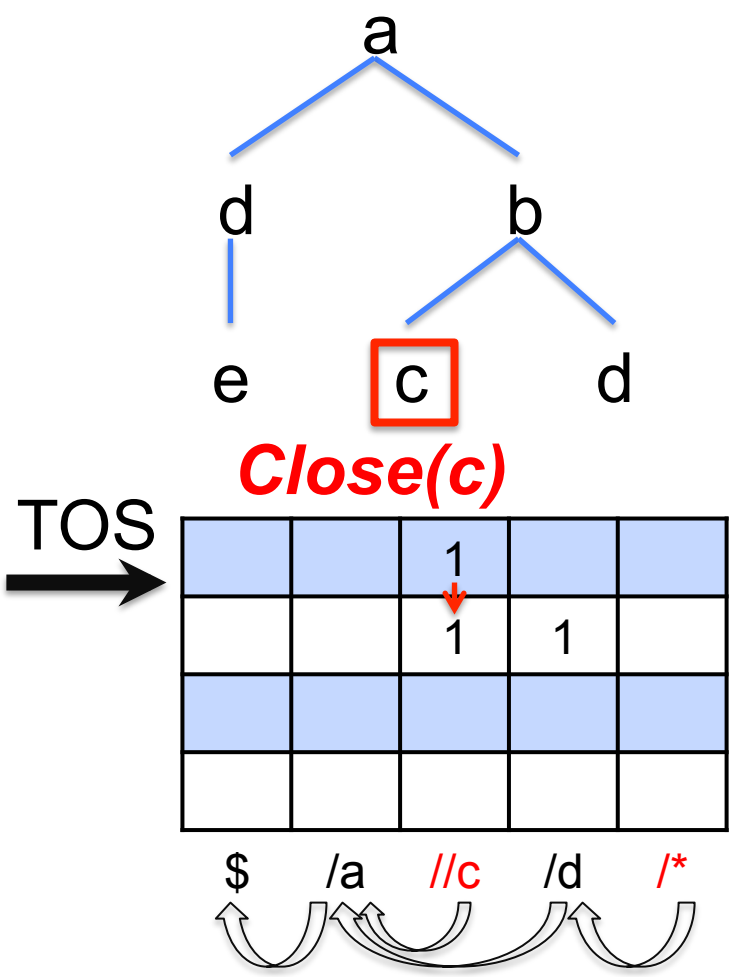




- Split node (‘/a’ in example) is matched only if **all** it’s children propagate ‘1’
- Since so far we have explored only one subtree of node ‘/a’ match cannot be propagated to this node (‘**and**’-logical operation used to match split node)

Pop stack: Example

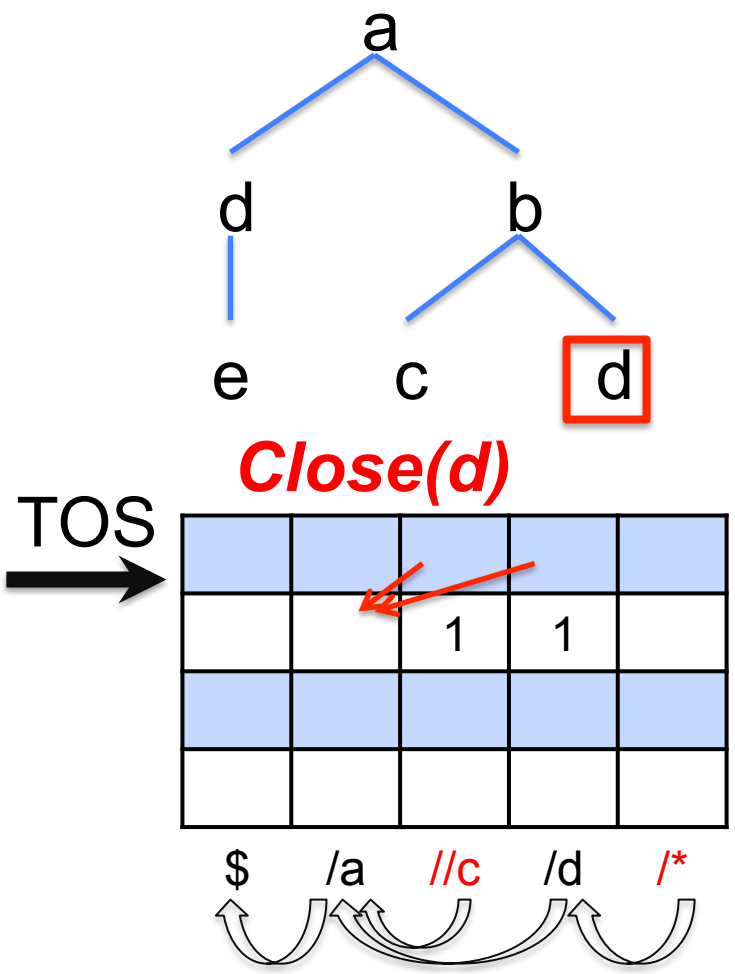
XML Document



- '1' is propagated downwards in descendant node if
  - Node holds '1' on TOS
  - Relationship with prefix is '//'
  - *Close* event tag matches column tag

Pop stack: Example

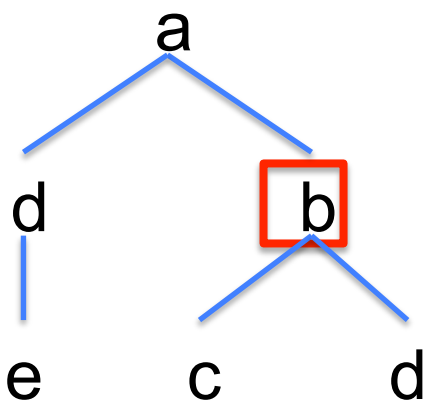
XML Document

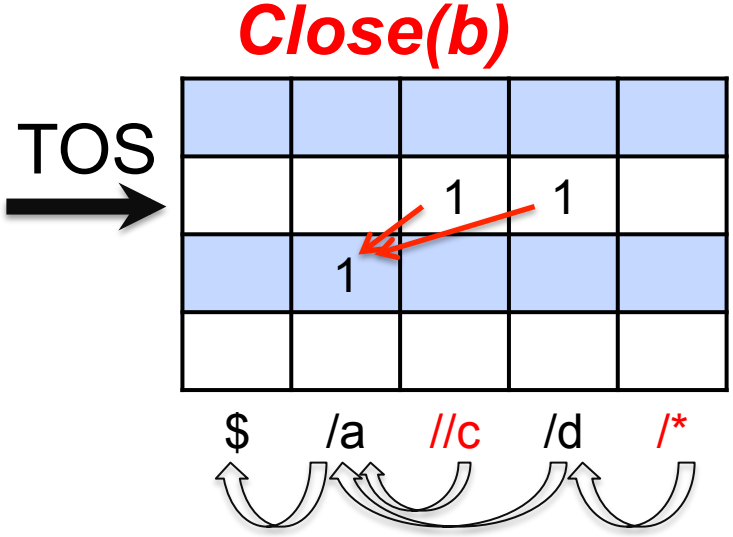


- The same rules apply, however nothing is propagated

Pop stack: Example

XML Document

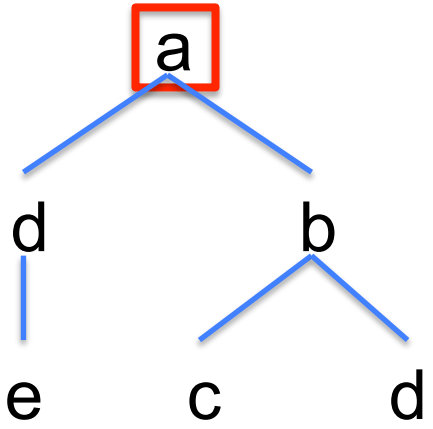




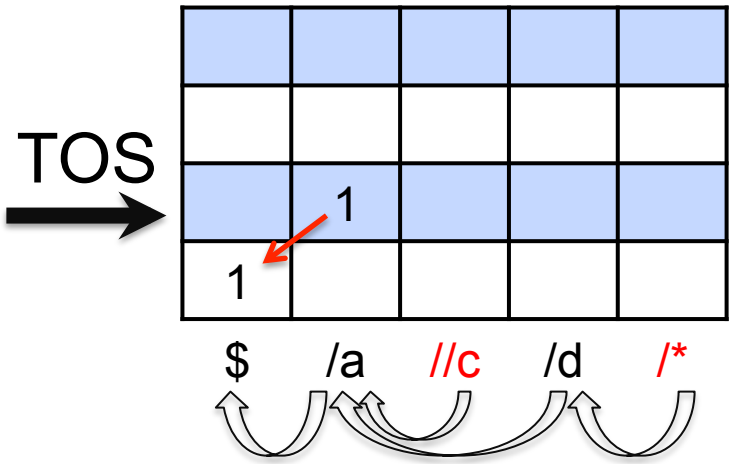
- This time both children of the split node are matched, thus the result of 'and'-operation propagates '1' down to node '/a'
- As with push stack split node has two separate fields for children with '/' and '//' relationships
- Reporting match from children to parent does not depend on event tag

Pop stack: Example

XML Document



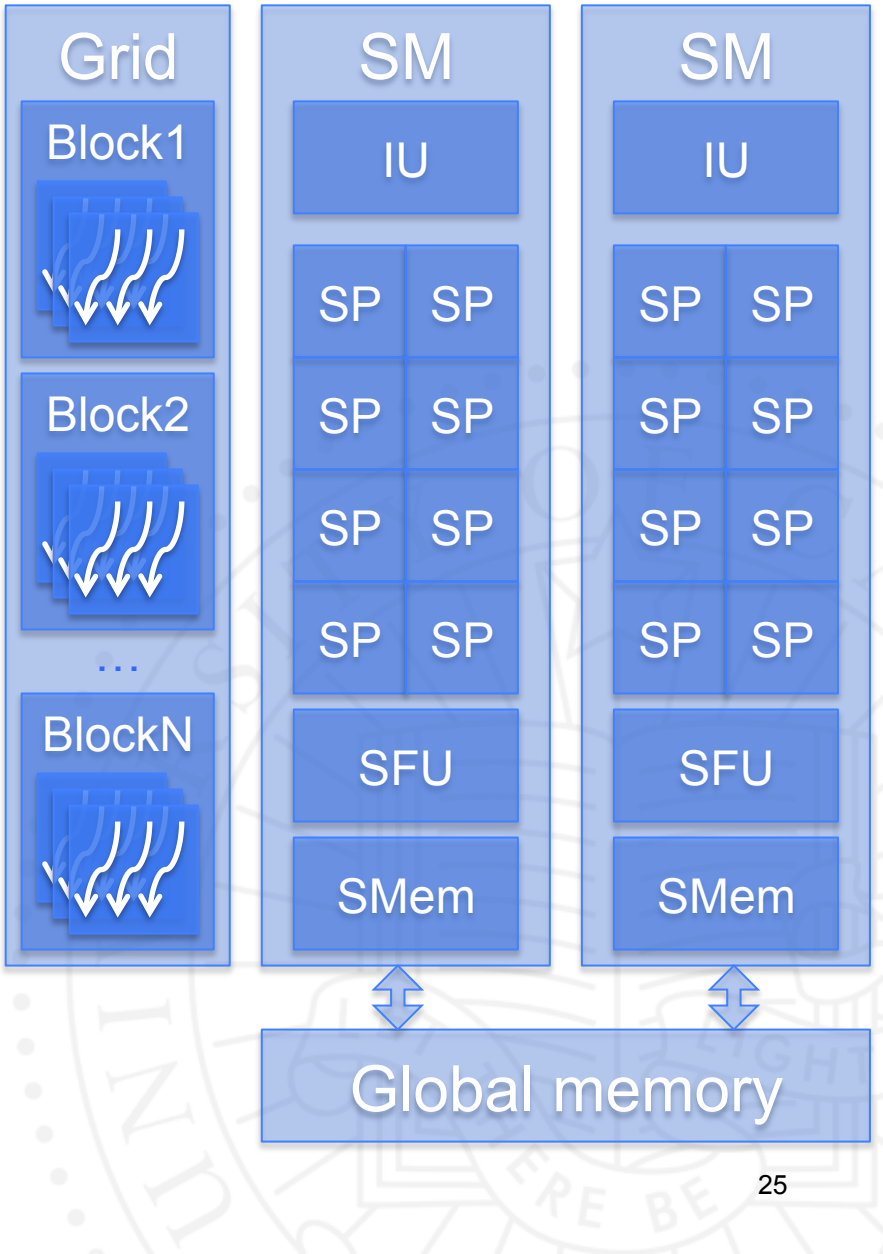
Close(a)



- Full query is matched if dummy root node reports match

GPU Architecture

- SM is a multicore processor, consisting of multiple SPs
- SPs execute the same instructions (kernel)
- SPs within SM communicate through small fast shared memory
- Block is a logical set of threads, scheduled on SPs within SM



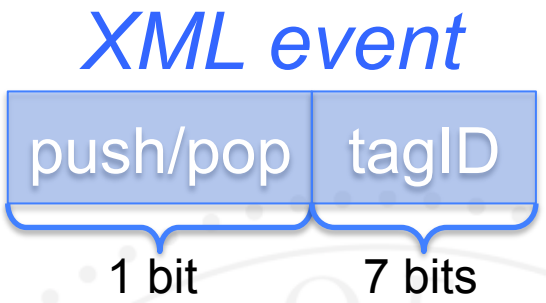
Filtering Parallelism on GPUs

- **Intra-query parallelism**
  - Each stack column on TOS is independently evaluated in parallel on SP
- **Inter-query parallelism**
  - Queries scheduled parallely on different SMs
- **Inter-document parallelism**
  - Filtering several XML documents as a time using concurrent GPU kernels (co-scheduling kernels with different input parameters)

XML Event & Stack Entry Encoding

- The XML document is preprocessed and transferred to the GPU as a stream of byte-long (event/ tag ID) pairs

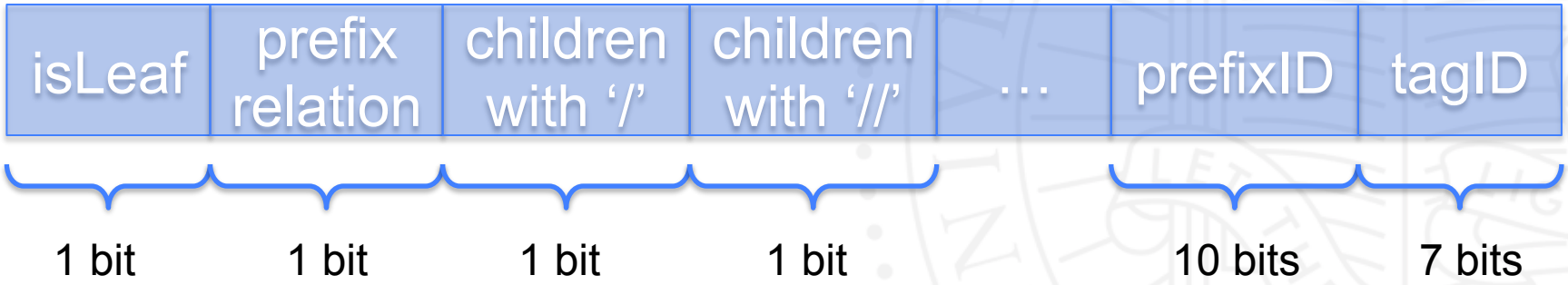
- The event streams reside in the GPU global memory



GPU Kernel Personality Encoding

- Each GPU kernel, maps to one query node
- The offline query parser creates a *personality* for each kernel, which is later stored in GPU registers
- A personality is a 4 byte entry

GPU personality



GPU Optimizations

- Merging push and pop stacks to save shared memory
- Coalescing global memory reads\writes
- Caching XML stream items in shared memory
  - Reading stream in chunks by looping in strided manner, since shared memory size is limited
- Avoiding usage of atomic functions
  - Calling non-atomic analogs in separate thread

Experimentation Setup

- GPU experiments
  - NVidia Tesla C2075(Fermi architecture), 448 cores
  - NVidia Tesla K20(Kepler architecture), 2496 cores

- Software filtering experiments
  - YFilter filtering engine
  - Dual 6-core 2.30GHz Intel Xeon E5 machine with 30 GB of RAM

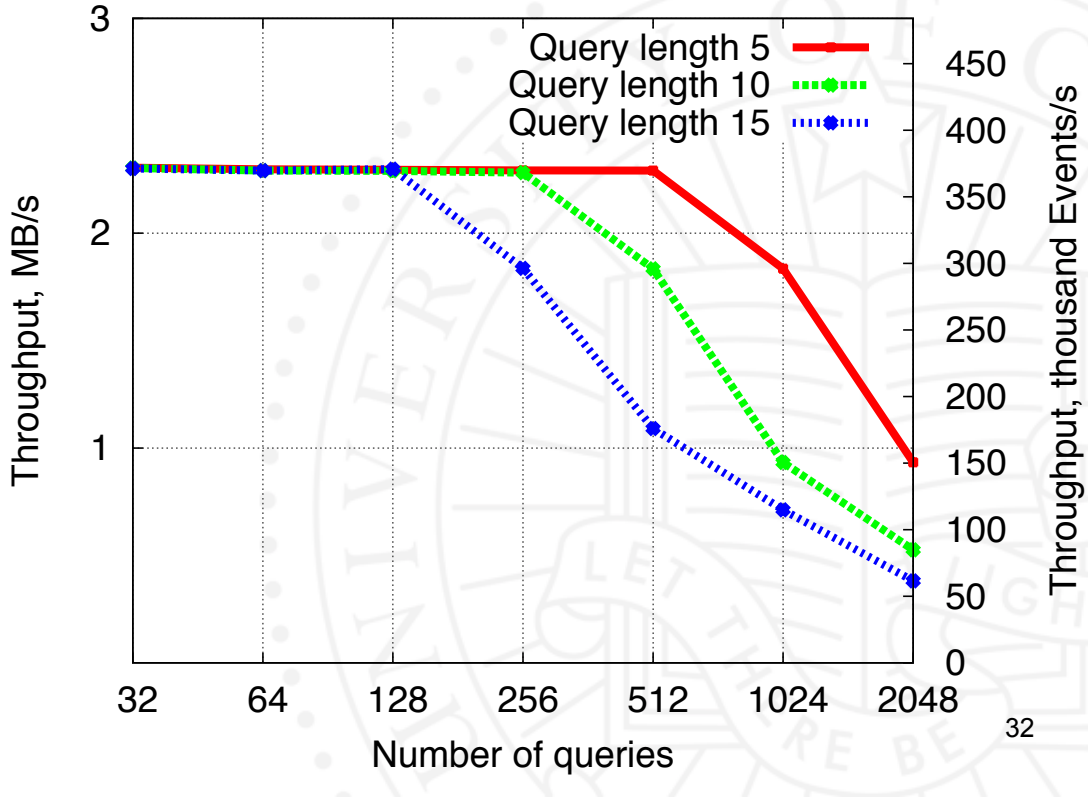
Experimentation Datasets

- DBLP XML dataset
  - Documents of varied size 32kB-2MB, obtained by trimming original DBLP XML
  - Synthetic documents (having DBLP schema) of fixed size 25kB
  - Maximum XML depth – 10

- Queries generated by the YFilter XPath generator with varied parameters
  - **Query size:** 5,10 and 15 nodes
  - **Number of split points:** 1,3 and 6
  - **Probability of '*' node and '//' relations:** 10%,30%,50%
  - **Number of queries:** 32-2k

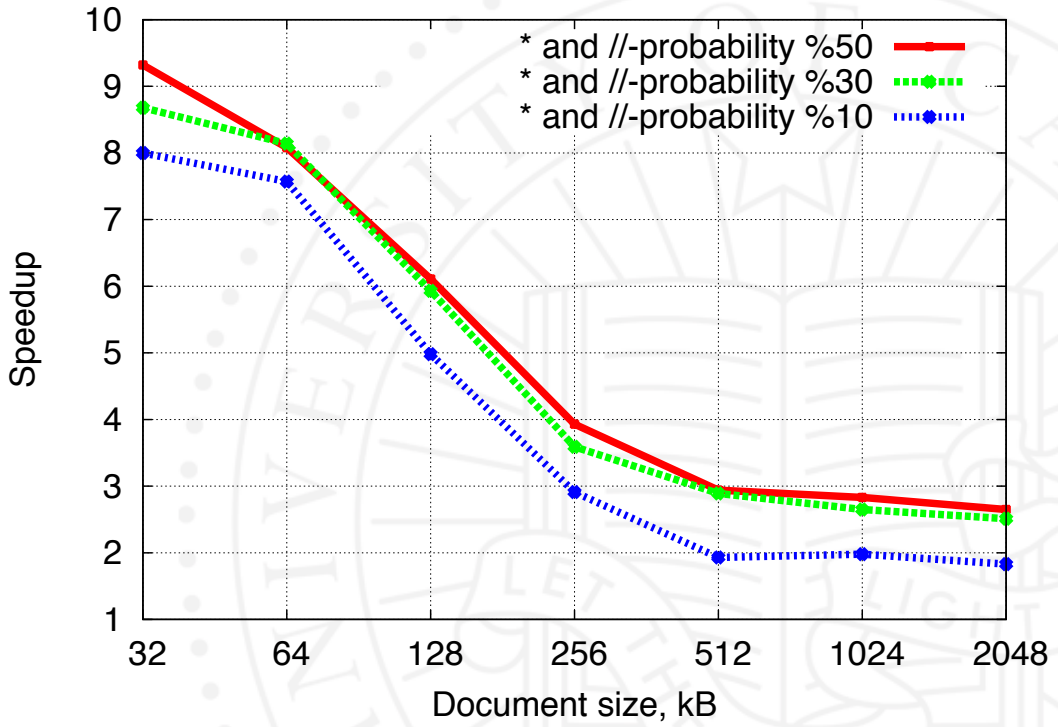
Experiment Results: Throughput

- GPU throughput is constant until “breaking” point – point where all GPU cores are occupied
- Number of occupied cores depends on number of queries and query length



Experiment Results: Speedup

- GPU speedup depends on XML document size: larger docs incur greater global memory read latency
- **Speedup up to 9x**
- '*' and '//'-probability affects speedup since it increases the YFilter NFA size

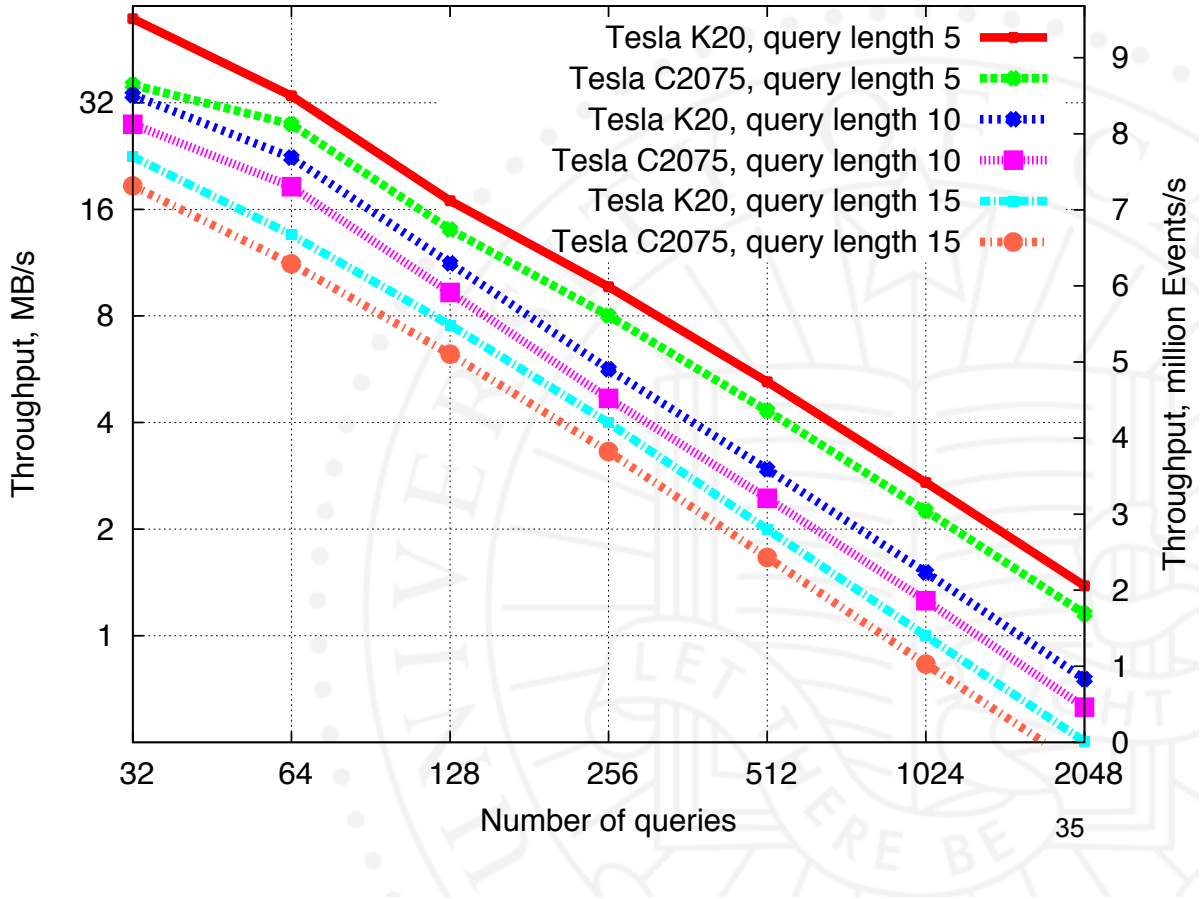


Batch Experiments

- Batched experiments show the usage of intra-document parallelism
  - **Mimic real case scenarios** (batches of real-time docs)
  - Batches of size 500 and 1000 docs were used
- It is nor fair to use single-threaded YFilter implementation in batch experiments
- “Pseudo”-multicore Yfilter: run multiple copies of program, distributing document load
  - Each copy runs on it’s own core
  - Each copy filters subset of document set, query set is fixed
  - Query load is the same for all copies, since it affect NFA size. Distributing queries deteriorates query sharing due to lack of commonalities.

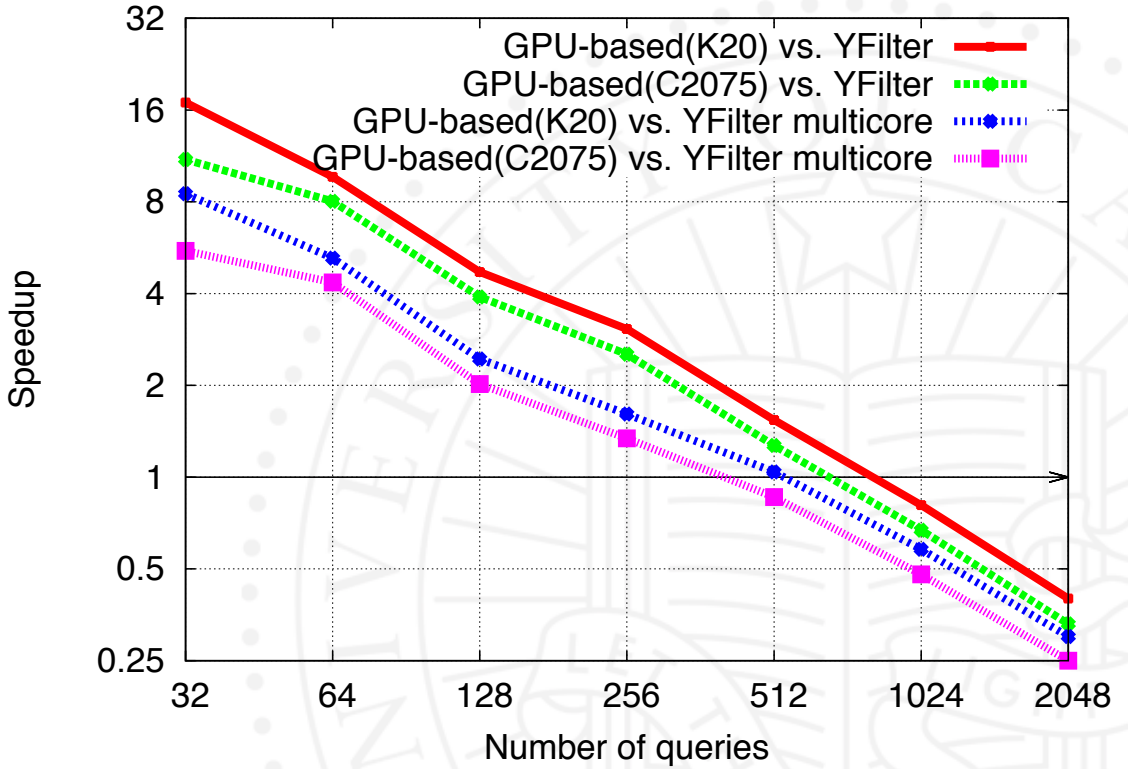
Batch Experiments: Throughput

- No breaking point – GPU is always fully occupied by concurrently executing kernels
- **Throughput increased up to 16 times** in comparison with single-document case



Batch Experiments: Speedup

- GPU fully utilized – increase in the query length number yields speedup drop by factor of 2
- Achieve up to 16x speedup
- **Slowdown** after 512 queries
- Multicore version performs better then ordinary



Conclusions

- Proposed the first holistic twig filtering using GPUs, effectively leveraging GPU parallelism
- Allow processing of thousands of queries, whilst allowing dynamic query updates (vs. FPGA)
- Up to 9x speedup over software systems in single-document experiments
- Up to 16x speedup over software systems in batches experiments

Thank you!

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE