High-Performance Holistic XML Twig Filtering Using GPUs

Ildar Absalyamov, Roger Moussalli, Walid Najjar and Vassilis Tsotras

UNIVERSITY OF CALIFORNIA, RIVERSIDE

Outline

- Motivation
- Related work
  - Software approaches
  - Hardware approaches
- Proposed approach & detailed algorithm
- Optimizations
- Experiments
- Conclusion

Motivation

- Filtering engine is the heart of pub-sub systems
  - Used to deliver news, blog updates, stock data, etc
- XML is standard format for data exchange
  - Powerful enough to capture message value as well as its structure using XPath
- Growing volume of information requires exploring massively parallel high-performance approaches for XML filtering

Related work (software)

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

FSM-based approaches

Related work (software)

- FiST (VLDB 2005)
  - Converts XML document into Prüfer sequences and matches respective sequences

sequence-based approaches

- XTrie (VLDB 2002)
  - Uses Trie-based index to match query prefix
- AFilter (VLDB 2006)
  - Leverages prefix as well as suffix query indexes

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
- “Massively parallel XML twig filtering using dynamic programming on FPGAs” (ICDE 2011)
  - Extended algorithm to support holistic twig filtering on FPGAs
- “Efficient XML path filtering using GPUs” (ADMS 2011)
  - Modified original approach to perform path filtering on GPUs

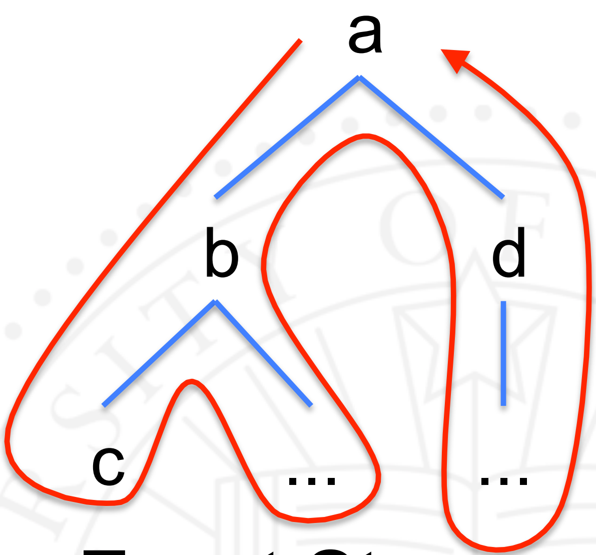
Why GPUs

- 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

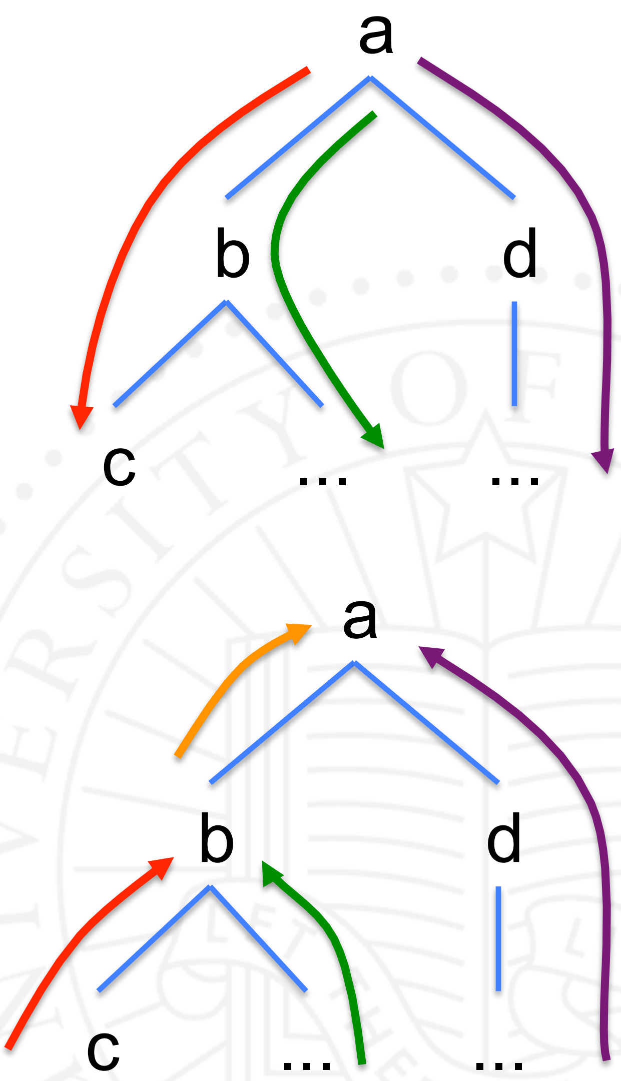


Event Stream

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

Twig Filtering: approach

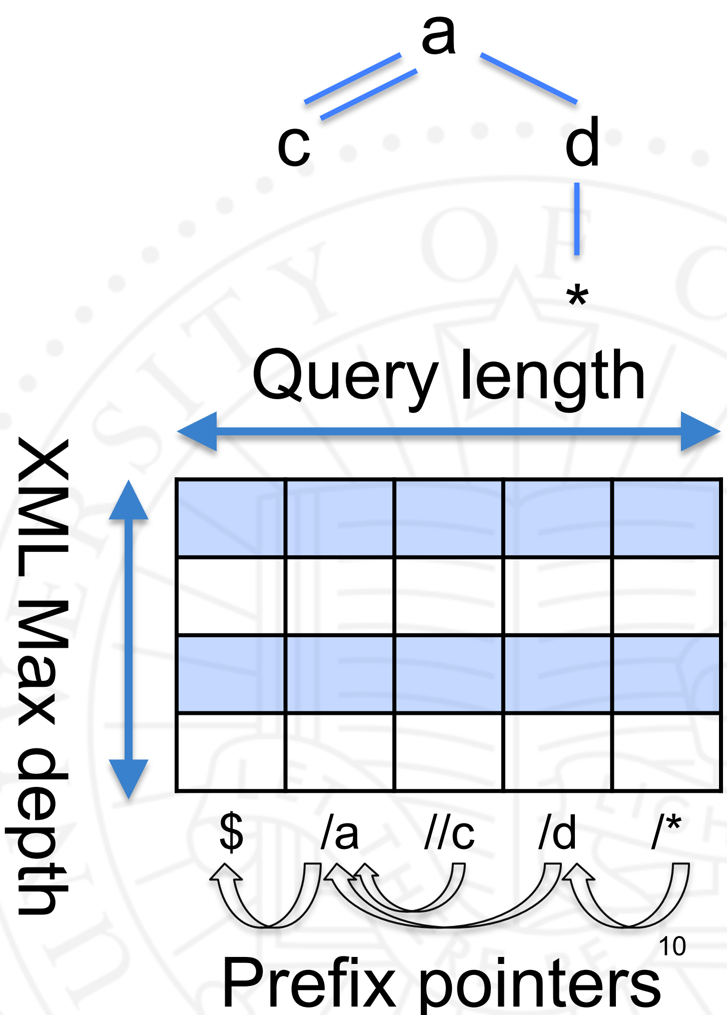
- Twig processing contains two steps
  - Matching individual root-to-leaf paths
  - Report matches back to root, while joining them at split nodes



Dynamic programming: algorithm

- Every query is mapped to DP table
- DP table - binary stack
- 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)

Query: /a[//c/]/d/*

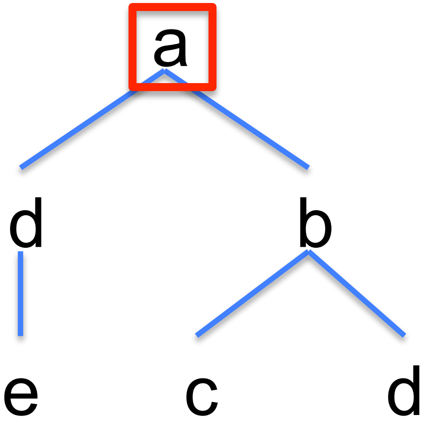


Dynamic programming: stacks

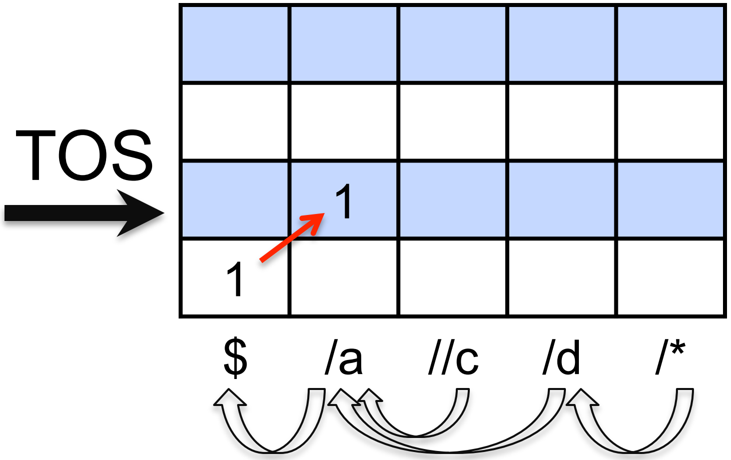
- Two different types of stack are used for different parts of filtering algorithm: push stack (for matching root-to-leaf paths) and pop stack (for propagating leaf matches back to root)
- TOS values of push stack are updated only during *open* events
- TOS values of pop stack are updated both on *open* and *close* events (overwrite existing information)

Push stack: Example

XML Document



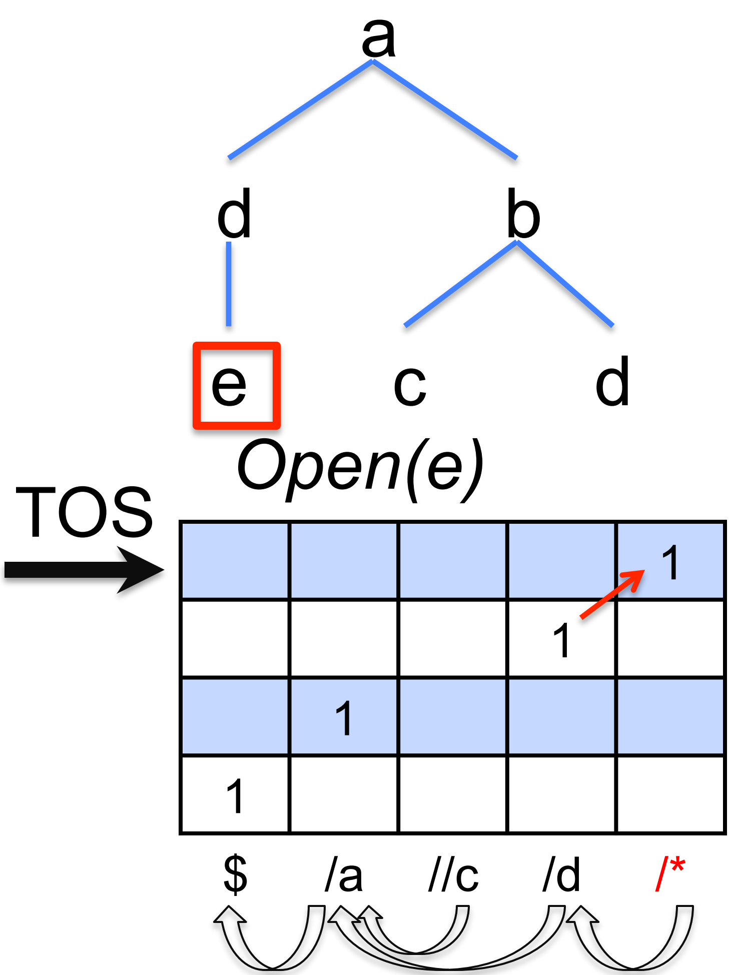
Open(a)



- 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

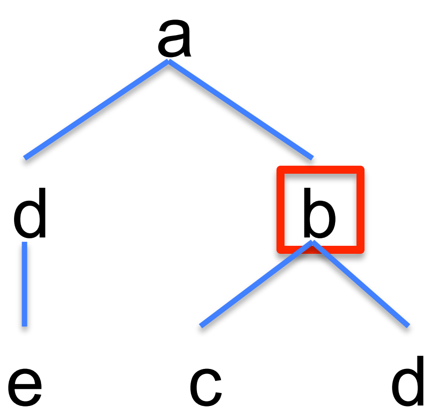


$	/a	//c	/d	/*
				1
			1	
	1			
1				

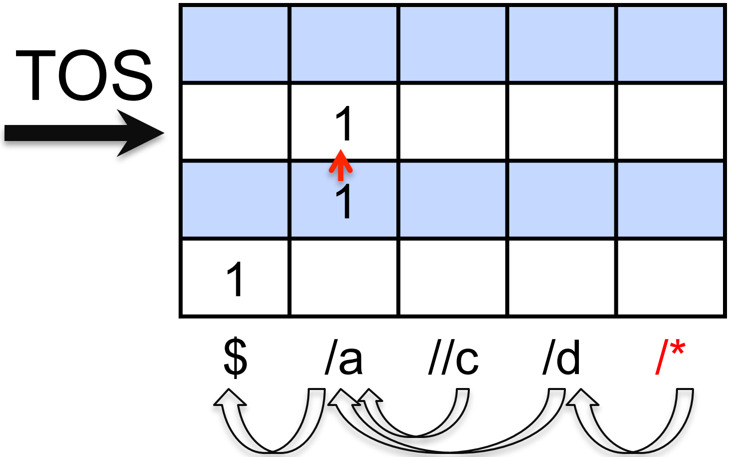
- If query node tag is wildcard (‘*’) then any tag in *open* event qualifies to be matched
- Since ‘/*’ is a leaf node matched this fact is saved in special binary array

Push stack: Example

XML Document



Open(b)

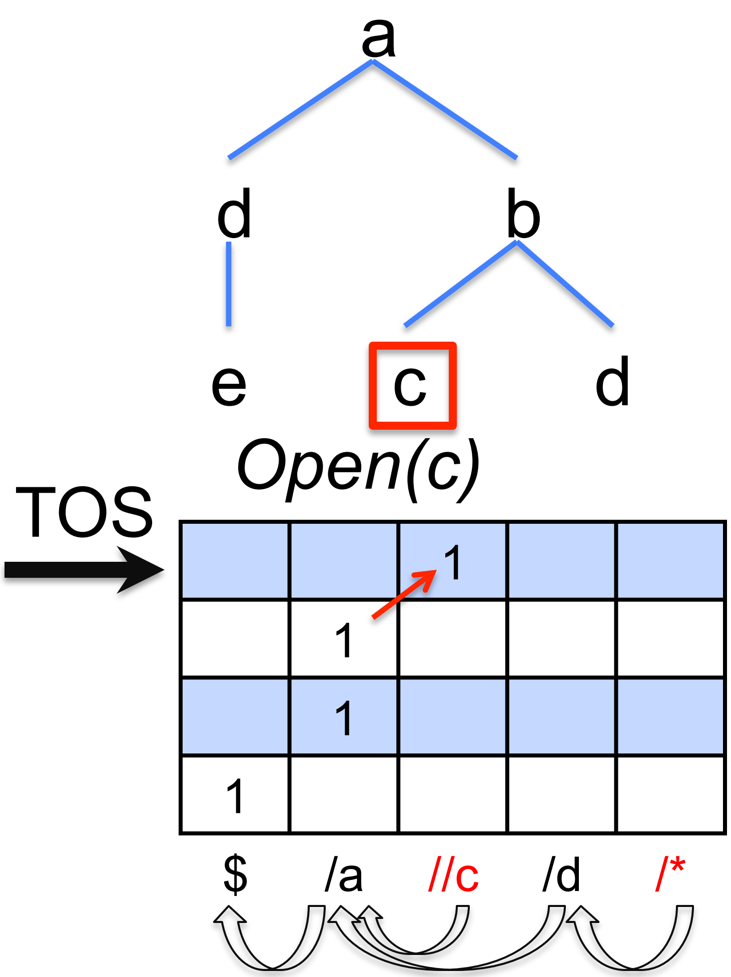


$	/a	//c	/d	/*
	1			
	1			
1				

- ‘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

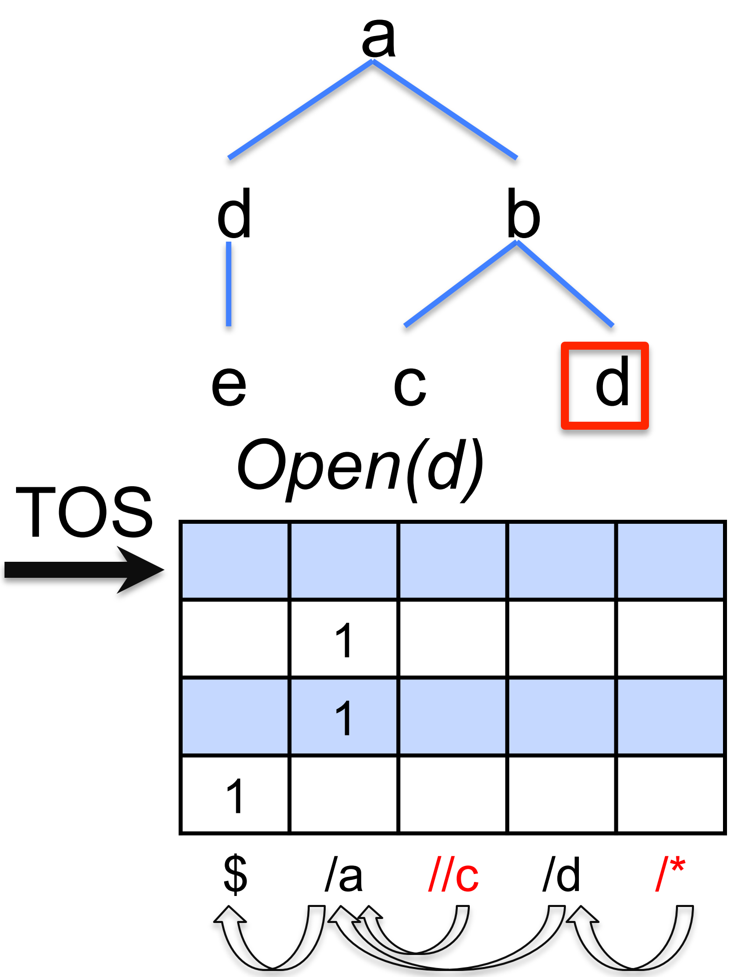


$	/a	//c	/d	/*
		1		
	1			
	1			
1				

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

Push stack: Example

XML Document

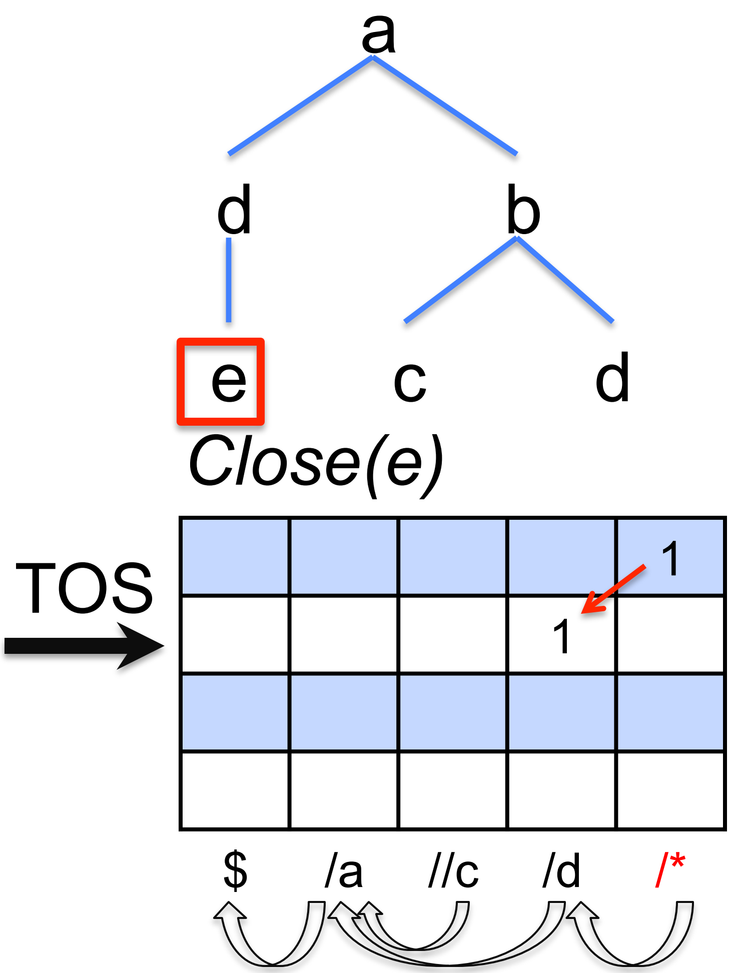


$	/a	//c	/d	/*
	1			
	1			
1				

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

Pop stack: Example

XML Document

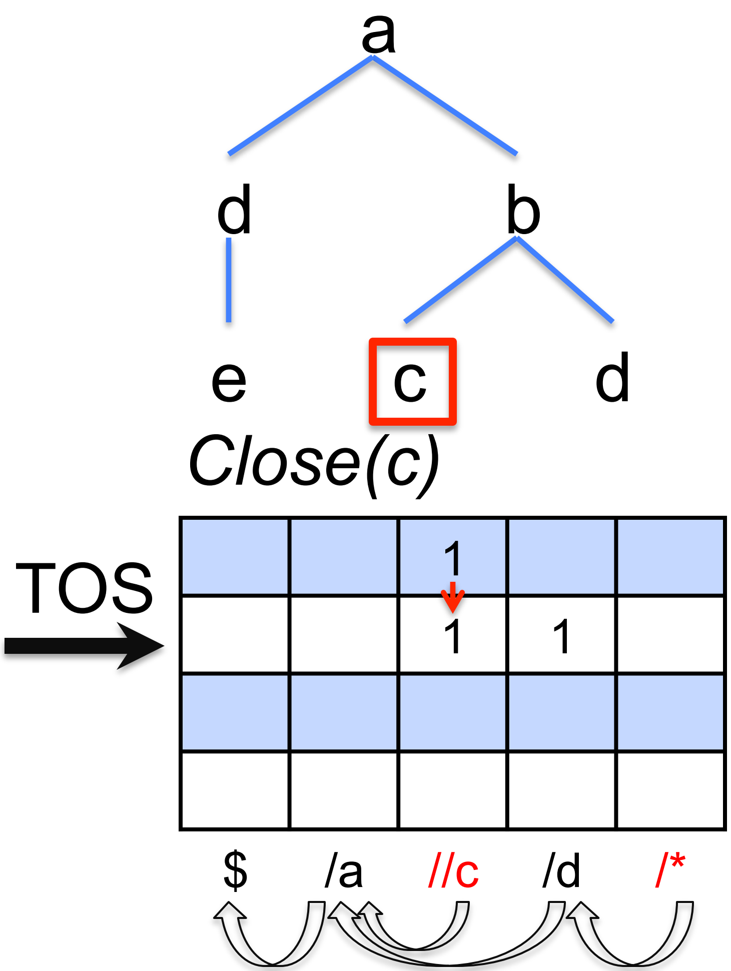


$	/a	//c	/d	/*
				1
			1	

- Leaf nodes contains ‘1’ if this node has saved in match node 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

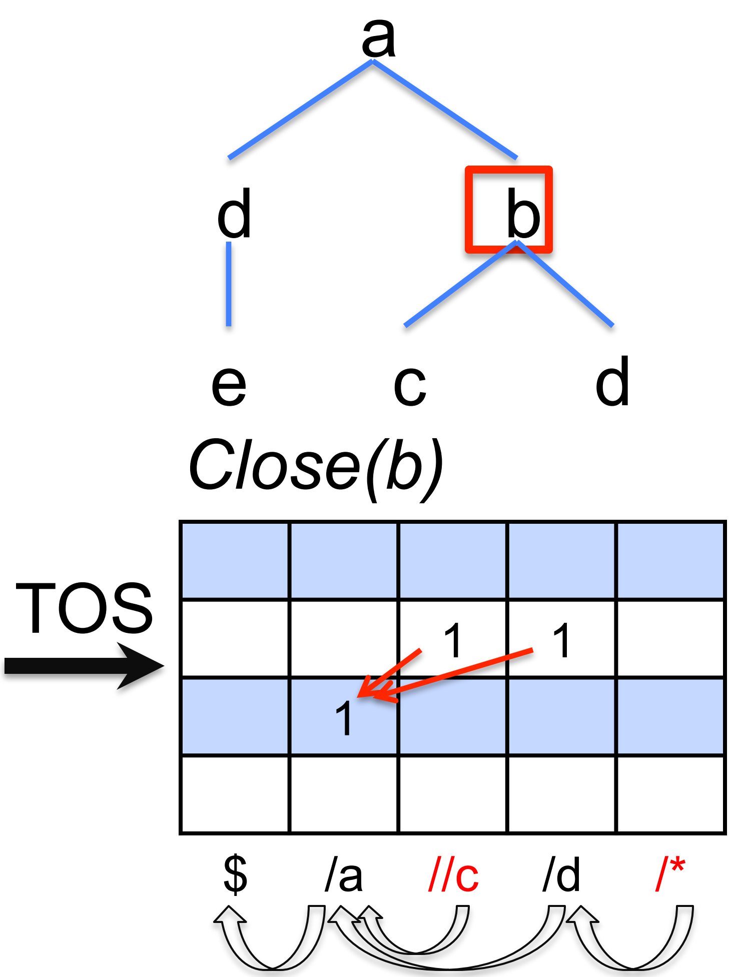


$	/a	//c	/d	/*
		1		
		1	1	

- ‘1’ is propagates 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

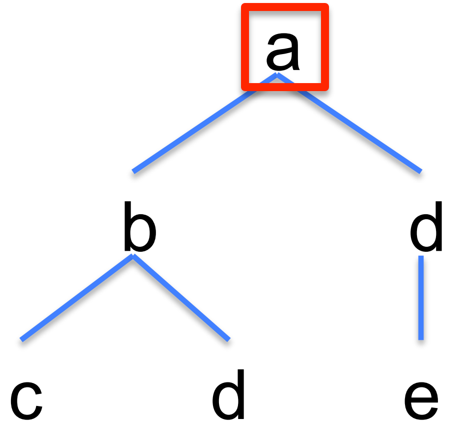


		1	1	
	1			
$	/a	//c	/d	/*

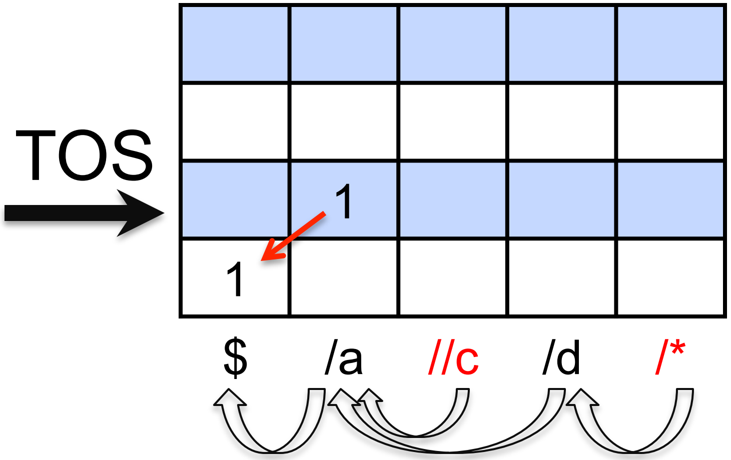
- Split node ('/a' in example) is matched only if **all** it's children propagate '1'
- As with push stack split node has two separate fields for children with '/' and '//' relationships
- Final match is obtained by **and**'ing these fields

Pop stack: Example

XML Document



Close(a)

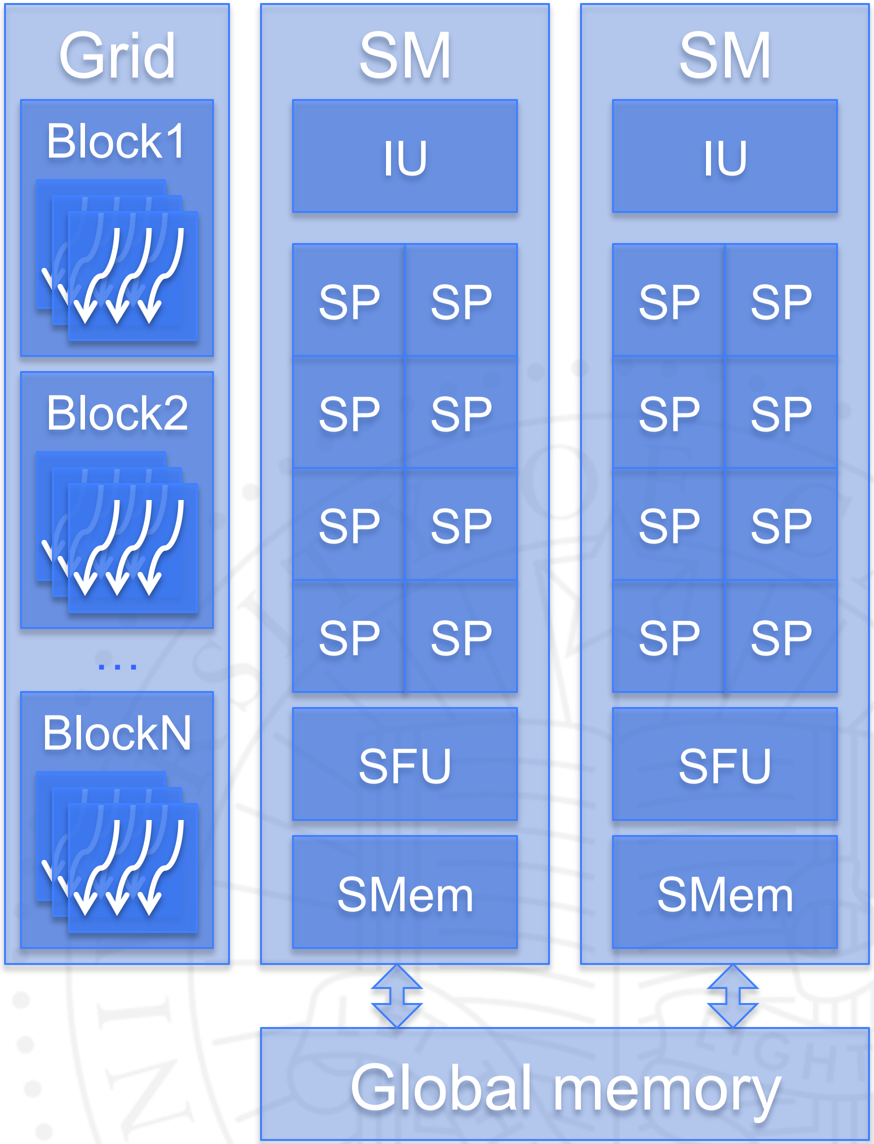


$	/a	//c	/d	/*
	1			
1				

- 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 Smem
- Block is a logical set of threads, scheduled on SPs within SM

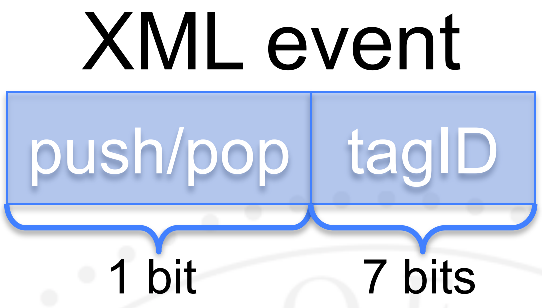


Filtering Parallelism on GPUs

- 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

- XML document is preprocessed and transferred to the GPU as a stream of byte-long events
- Event streams reside in global memory



GPU Kernel Personality Encoding

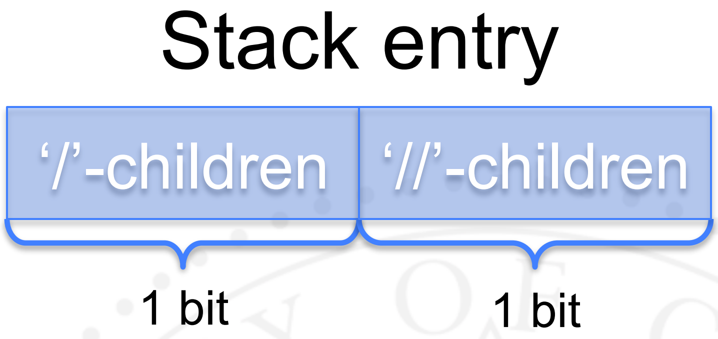
- Each GPU kernel, maps to one query node
- Kernel receives the description of this node as an input parameter, called *personality*
- Query parser creates personalities
- Once personality is received it is stored GPU registers

GPU personality

isLeaf	prefix relation	children with '/'	children with '//'	…	prefixID	tagID
1 bit	1 bit	1 bit	1 bit		10 bits	7 bits

Stack entry Encoding

- To address semantics of the split node, having children with different types of relationship we need to have 2 fields within stack entry
- Stacks reside in shared memory


Stack entry

‘/’-children	‘//’-children
1 bit	1 bit

GPU Optimizations

- Physically 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 XML stream cannot be placed in shared memory as a whole
- Avoiding usage of atomic functions
  - Calling non-atomic analogs in separate thread

Experiment Setup

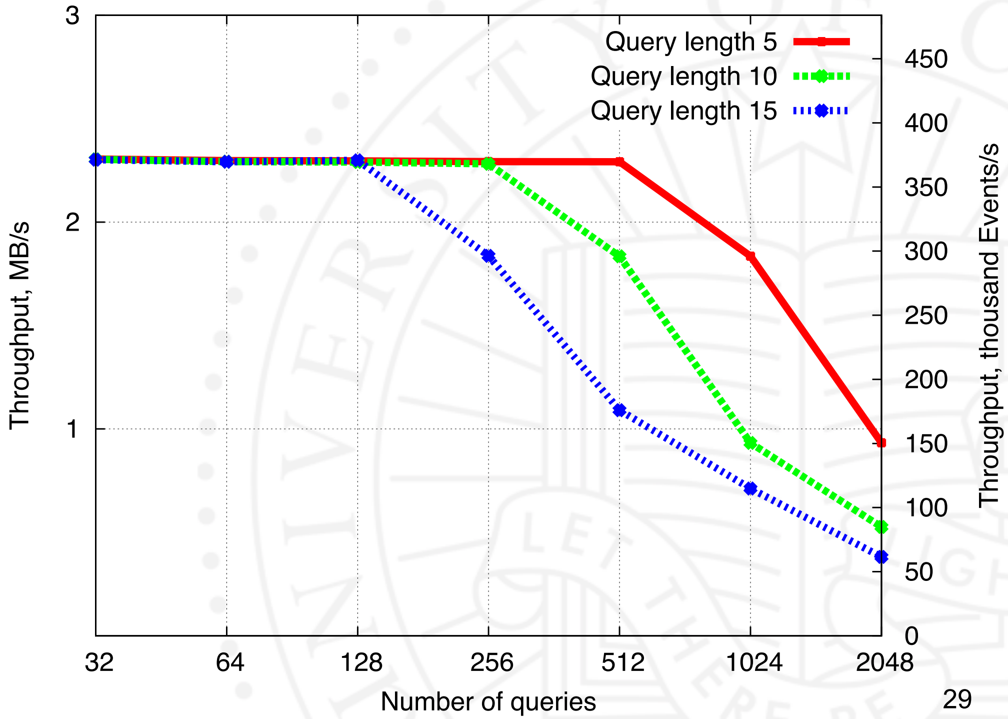
- GPU experiments
  - NVidia Tesla C2075(Fermi architecture),448 cores
  - NVidia Tesla K20(Kepler architecture),2496 cores
- Software filtering experiments
  - YFilter filtering engine
  - Dual 6-core 2.30GHz Intel Xeon E5 machine with 30 GB of RAM

Experiment Datasets

- DBLP XML dataset
  - Chunks of varied size 32kB-2MB from original dataset
  - Synthetic documents of size 25kB
  - Maximum XML depth - 10
- Queries, generated by 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 '//' relation 10%,30%,50%
  - Number of queries 32-2k

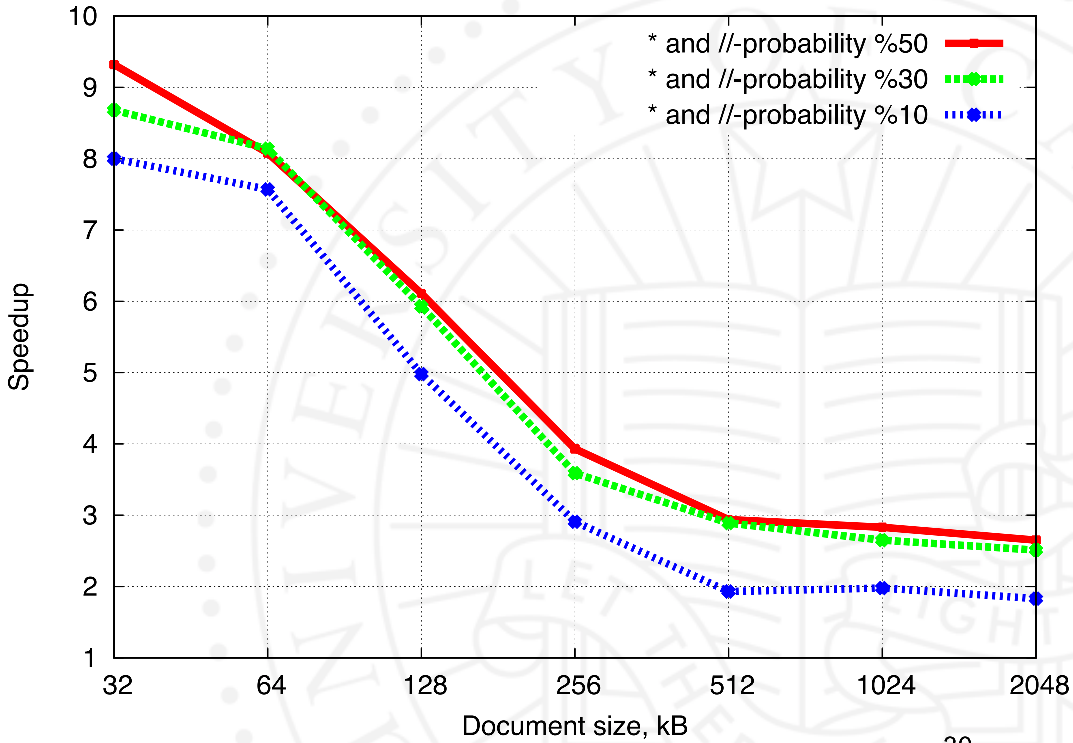
Experiment Results: Throughput

- GPU throughput (for 1MB document) 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 YFilter NFA size



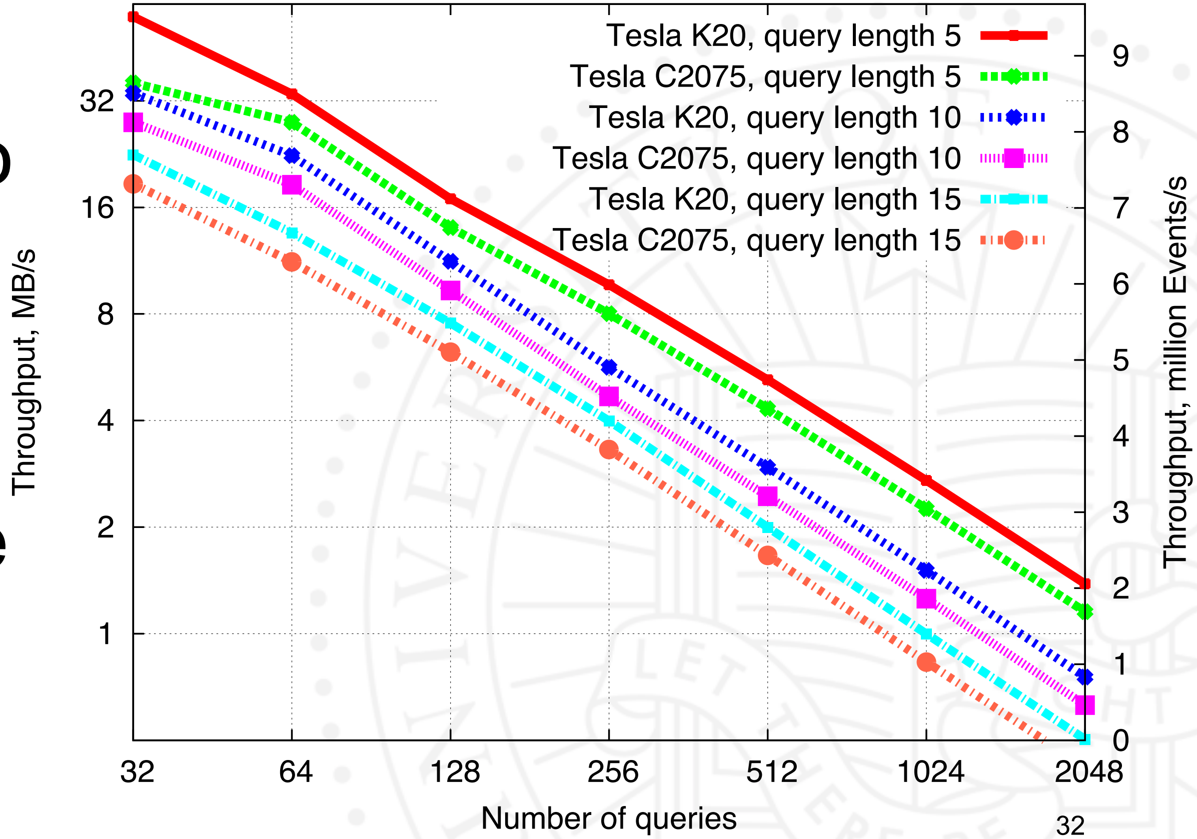
Batch Experiments

- Batched experiments filter multiple XML documents
  - Shows usage of intra-document parallelism
  - Batches of size 500 and 1000 were used
- It is nor fair to compare against single-threaded Yfilter in batch experiments
- “Pseudo”-multicore YFilter version: distributes document load across different copies of program
  - Could not be done for query load, would affect NFA size

Batch Experiments: Throughput

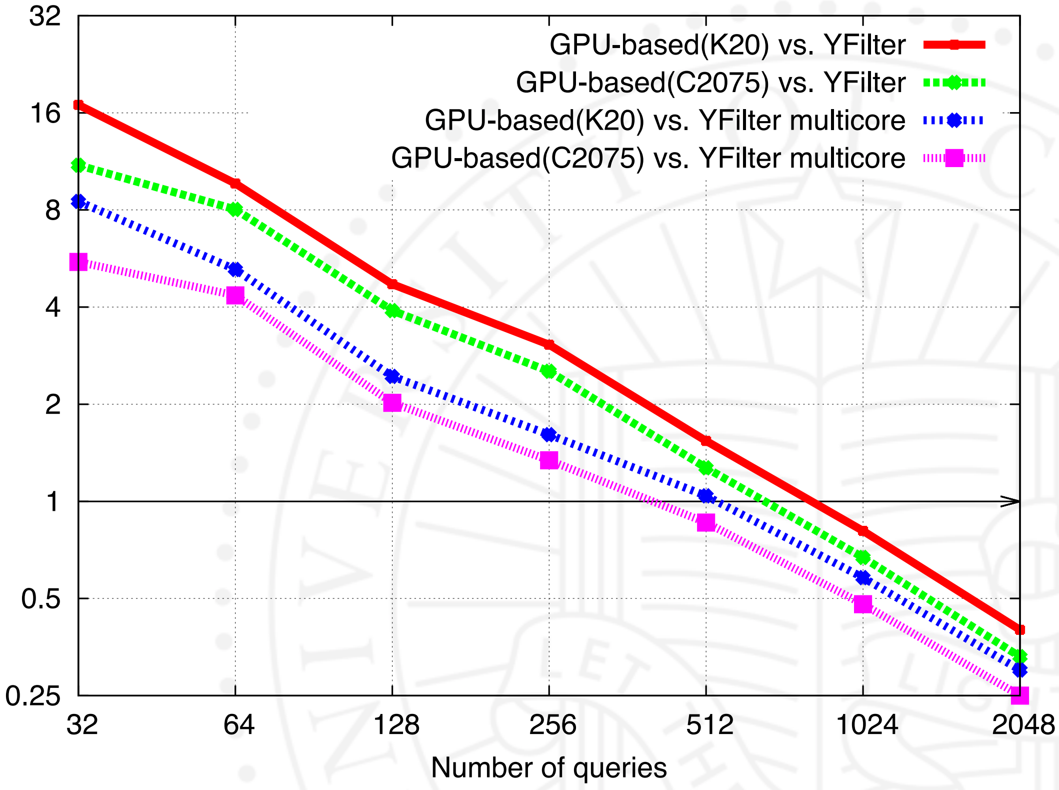
- No breaking point – GPU is always fully occupied by concurrently executing kernels

- Throughput increased up to 16 times in comparison with single-document case



Batch Experiments: Speedup

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



Conclusions

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

Thank you!