Efficient XML Path Filtering Using GPUs

Roger Moussalli, Robert Halstead, Mariam Salloum,
Walid Najjar and Vassilis Tsotras

University of California, Riverside

Motivation

- XML has become the de-facto standard for data distribution and exchange.
  - News feeds, scientific data, stocks, etc.

- An important problem in XML is the *filtering algorithm:*
  - For each XML document, find the set of queries that have at least 1 match in the document.

- With the growing amount of distributed information, current software-based approaches are unable to handle a large volume of input stream.

Outline

- Related Work
- Path Matching
    - Algorithm
    - Path Matching Example
- System Architecture
- Experiments
- Conclusions

Related Works (sw)

- **Xfilter** (*VLDB 2000*)
  - Transform each query into an FSM.
- **YFilter** (*TODS 2003*)
  - Decompose twigs into paths, and generate a single NFA to represent the set of queries.
- **LazyDFA** (*TODS 2004*)
  - Build deterministic FSMs.

FSM-based approaches

- **FiST** (*VLDB 2005*)
  - XML filtering through the sequencing of profiles and of the XML document.

Sequence-based approaches

- **Afilter** (*VLDB 2006*)
- **XTrie** (*ICDE 2002*)
- **Relational XML pub-subs** (*SIGMOD 2004*)

Others

Related Works (hw)

- XML Parsing
  - “**XML Accelerator Engine**” (*High Performance XML Processing 2004*). Parser handles 1 byte of data per cycle.
  - “**A 1 cycle-per-byte XML Parsing Accelerator**” (*FPGA 2010*). Parser handles 2 to 4 bytes of data per cycle; support for XML Schema validation is also offered.

Related Works – XML Filtering (hw)

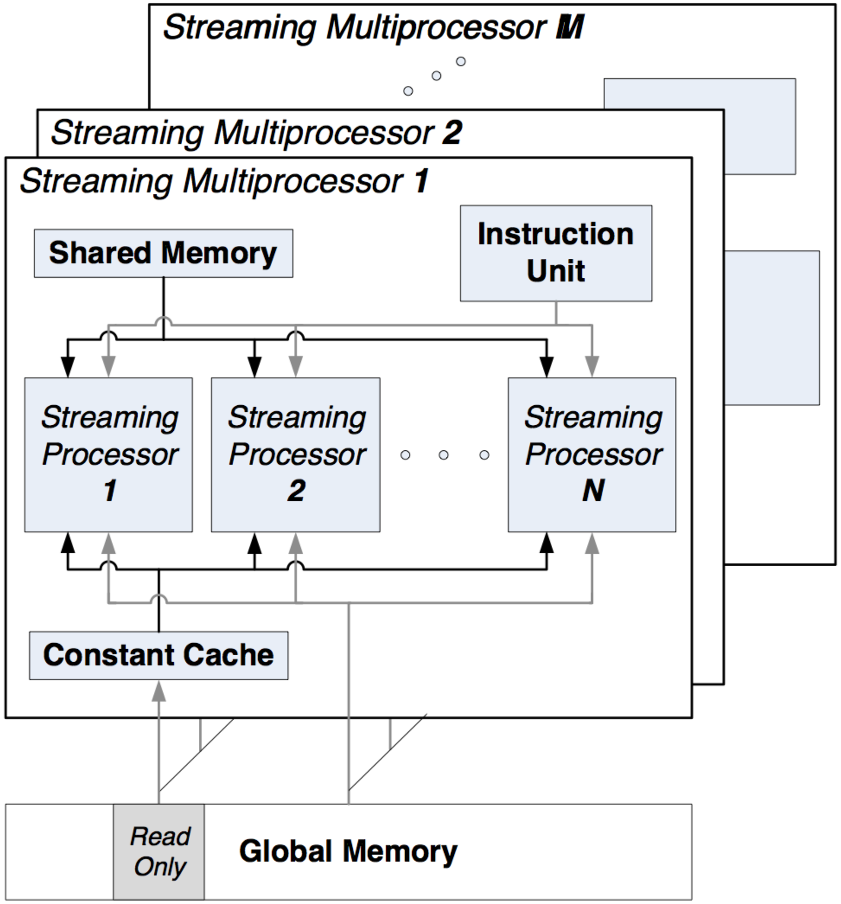
- XML Filtering
  - **“Boosting XML through a scalable FPGA-based architecture”** (*CIDR 2009*)
    - Every query is mapped onto a hw NFA.
    - Assumes a pre-processed XML document.
    - No support for recursion, wildcards.
    - Simultaneous matches are not handled.
  - **“Accelerating XML Query Matching Through Custom Stack Generation on FPGAs”** (*HiPEAC 2010*)
    - Queries are mapped onto custom binary stacks.
    - Highly parallel framework.
    - High throughput (200MB/s).
    - Up to *three orders of magnitude* speedup vs software approaches.
  - **“Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs”** (*ICDE 2011*)
    - Queries are more complex (twigs vs paths).
    - Fewer queries can fit on an FPGA.

Why GPUs

- **Limitations of FPGAs**:
  - Current generations of FPGA technology are **limited in resources** (limit at 8K queries).
  - **Query updates** incur a lengthy process:
    - Re-generating hardware.
    - Synthesis/Place & Route.
- **GPUs as co-processors**:
  - **Highly parallel** architectures, suitable for SIMD-type applications.
  - Operate at **high frequencies** (1GHz vs 200MHz).
  - Provide the **flexibility** of software.
  - **Updating** the functionality of the GPU is a fast process:
    - Minimal query update time.
  - Our proposed FPGA-based filtering algorithm can be further extended, being a potential good fit for GPUs (thousands of simple operations in parallel).

Overview of GPUs

- **Streaming processors** (SPs) are clustered into **streaming multiprocessors** (SMs).
- SPs within an SM communicate through a fast, small **shared memory**.
- SPs across SMs communicate through the high latency **global memory**.
- All SPs execute the same set of instructions (**kernel**).
- The group of kernels actively executing on an SM at a time is referred to as the **block**.

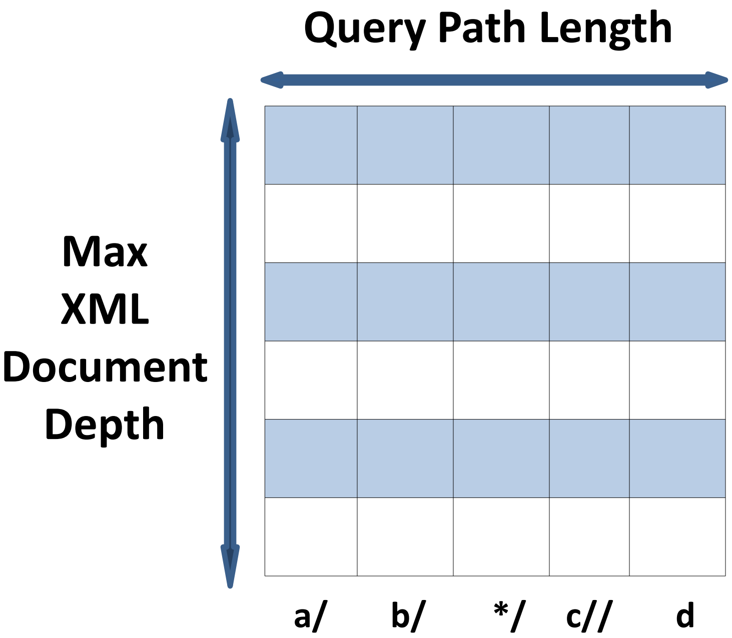


Proposed Filtering Approach

Proposed Algorithm

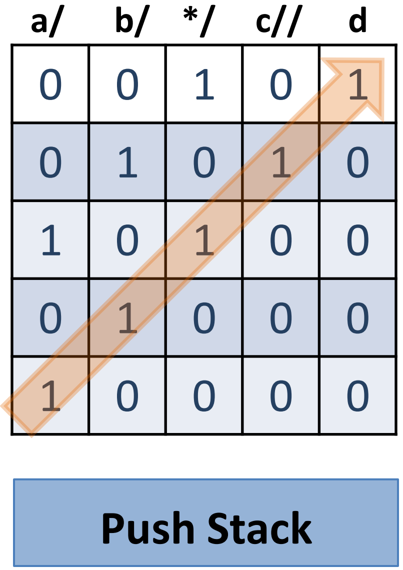
- Dynamic programming approach:
  - Every path is mapped to a *dynamic table*.
  - Every *node* in the path query is mapped to a stack *column*.
  - The dynamic table is a binary stack.
  - ***Open(tag)*** and ***close(tag)*** XML events translate into **push** and **pop** actions on the stack, respectively.
- Exploited parallelism:
  - **Inter-query parallelism –** query matching engines (dynamic tables) operate in parallel.
  - **Intra-query parallelism –** Query nodes (columns) are updated in parallel (top of stack).

Path Query: /a/b/*/c//d



Proposed Algorithm (cont'd)

- Inter-Column Relations (applied on push events):
  - 1's propagate diagonally upwards from left to right, in adjacent columns.
  - Custom checks have to be made when propagating a '1', based on the relation between respective two nodes.
  - Matching a path of length N requires the matching of the sub-path of length N-1.
  - Upon a match, a '1' would have propagated diagonally from the first to the last column.
  - Recursion is supported since multiple partial matches can occur simultaneously.

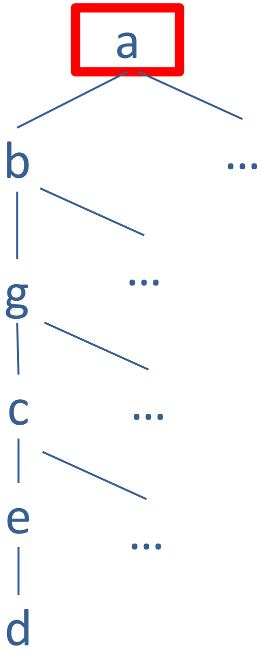


a/	b/	*/	c//	d
0	0	1	0	1
0	1	0	1	0
1	0	1	0	0
0	1	0	0	0
1	0	0	0	0

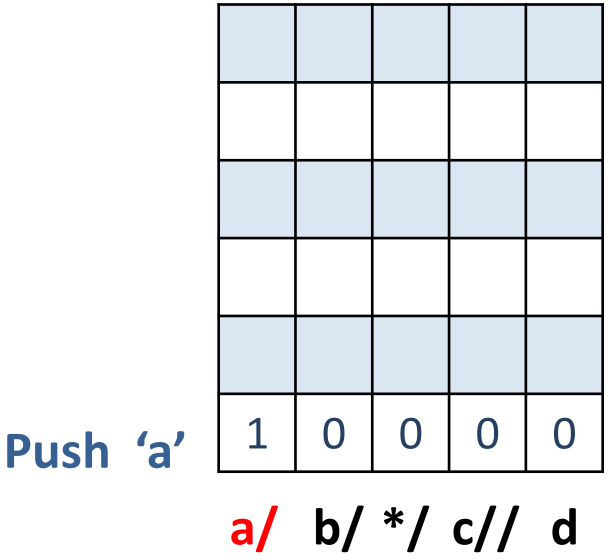
Push Stack

Path Matching Walk-Through

Path Matching Walk-Through

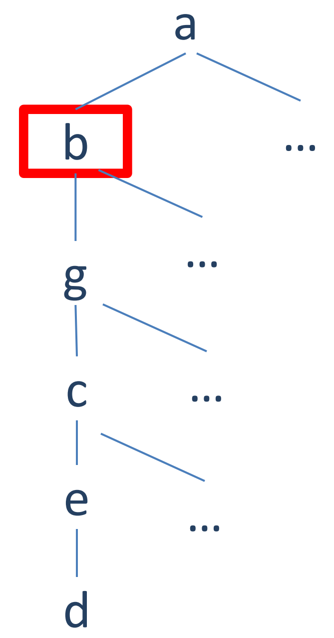


XML Tree



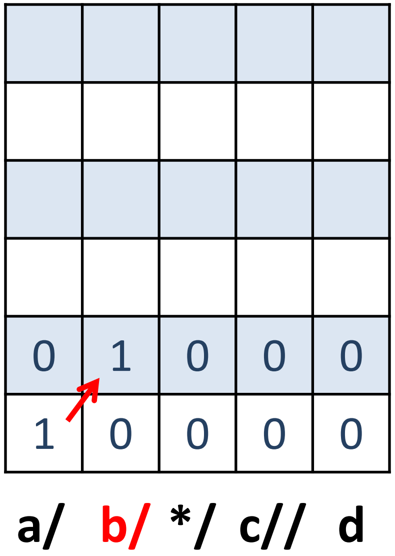
Query Stack

Path Matching Walk-Through



XML Tree

Push ‘b’



a/	**b/**	*/	c//	d
0	1	0	0	0
1	0	0	0	0

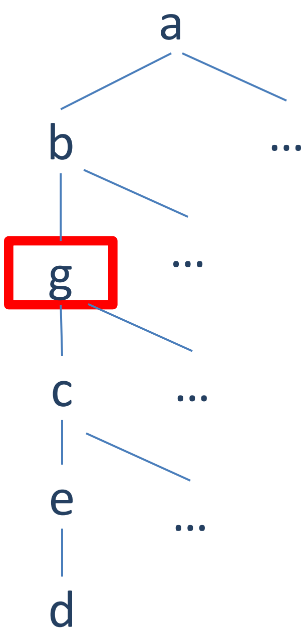
Query Stack

At the new top of stack of every column:

A ‘1’ propagates **diagonally upward** if:

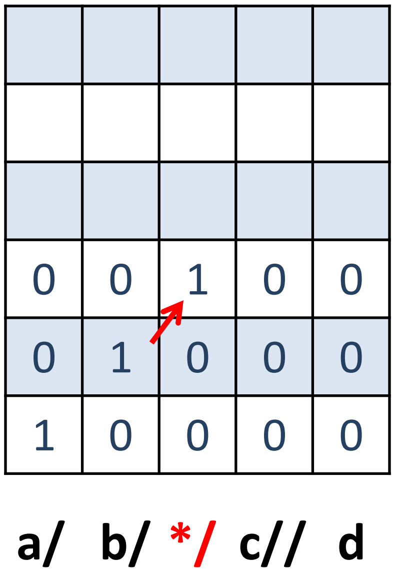
1. The previous column holds a ‘1’ at the earlier top of stack (‘a/’ column).
2. The pushed XML element matches the current column tag (here ‘b/’).

Path Matching Walk-Through



XML Tree

Push ‘g’



a/	b/	*/	c//	d
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0

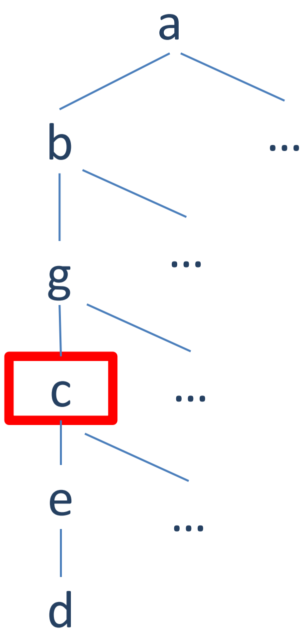
Query Stack

At the new top of stack of every column:

A ‘1’ propagates **diagonally upward** if:

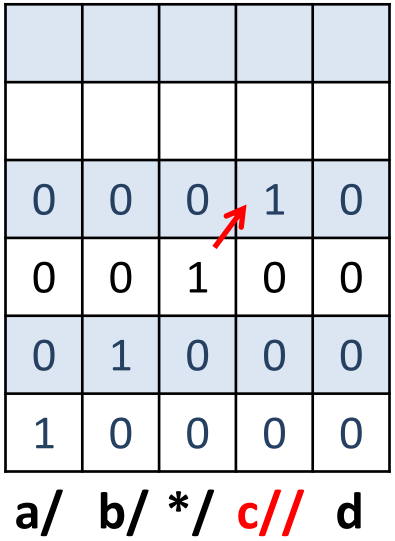
1. The previous column holds a ‘1’ at the earlier top of stack (‘b/’ column)
2. A wildcard is mapped to the current column.

Path Matching Walk-Through



XML Tree

Push ‘c’



a/	b/	*/	c//	d
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0

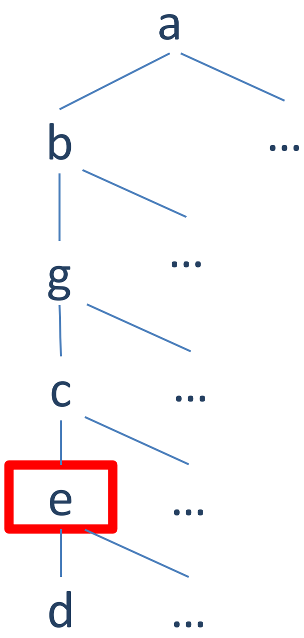
Query Stack

At the new top of stack of every column:

A ‘1’ propagates ***diagonally upward*** if:

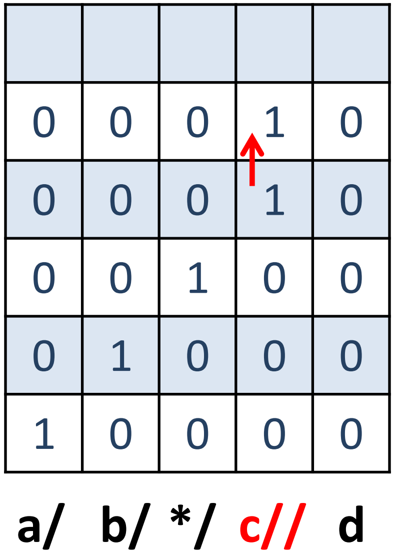
1. The previous column holds a ‘1’ at the earlier top of stack (‘*/’ column).
2. The pushed XML element matches the current column tag (here ‘c’).

Path Matching Walk-Through



XML Tree

Push ‘e’



0	0	0	1	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0
a/	**b/**	***/**	**c//**	**d**

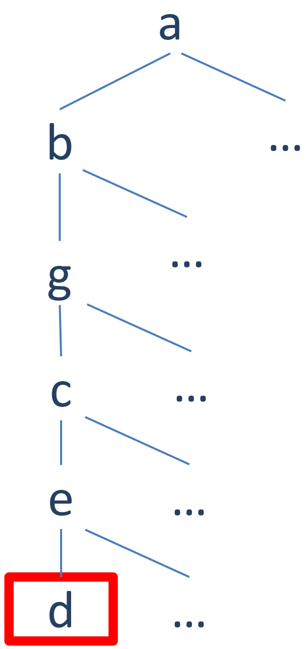
Query Stack

At the new top of stack of every column:

A ‘1’ propagates ***vertically upward*** if :

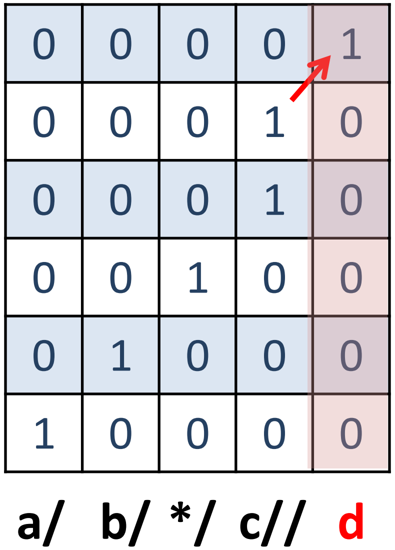
1. The column holds a ‘1’ (here ‘c//’).
2. The column has a ‘//’ relationship (here ‘c//’).

Path Matching Walk-Through



XML Tree

Push 'd'



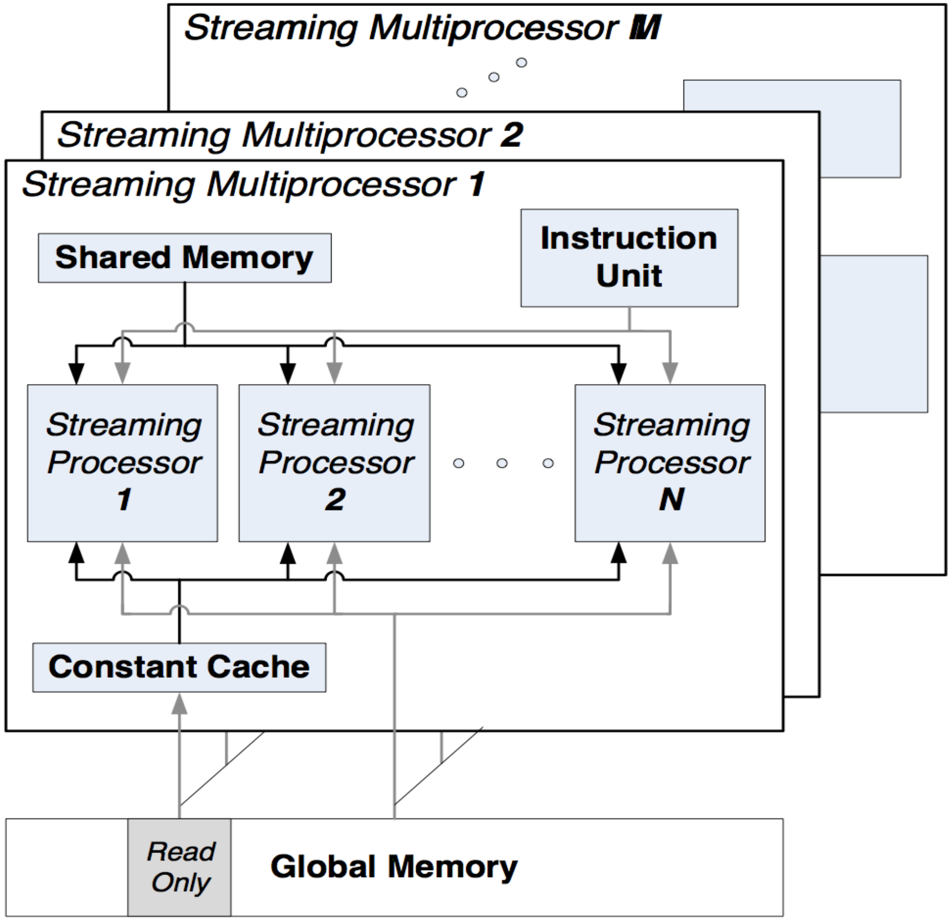
a/	b/	*/	c//	d
0	0	0	0	1
0	0	0	1	0
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	0

Query Stack

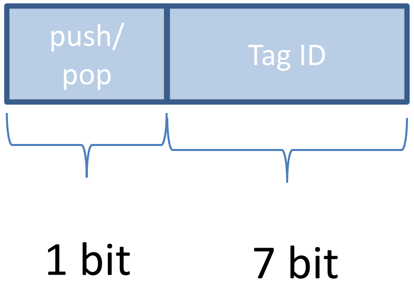
A '1' in the last column (leaf node column) **indicates a match.**

Path Filtering as Applied to GPUs

Path Filtering on GPUs

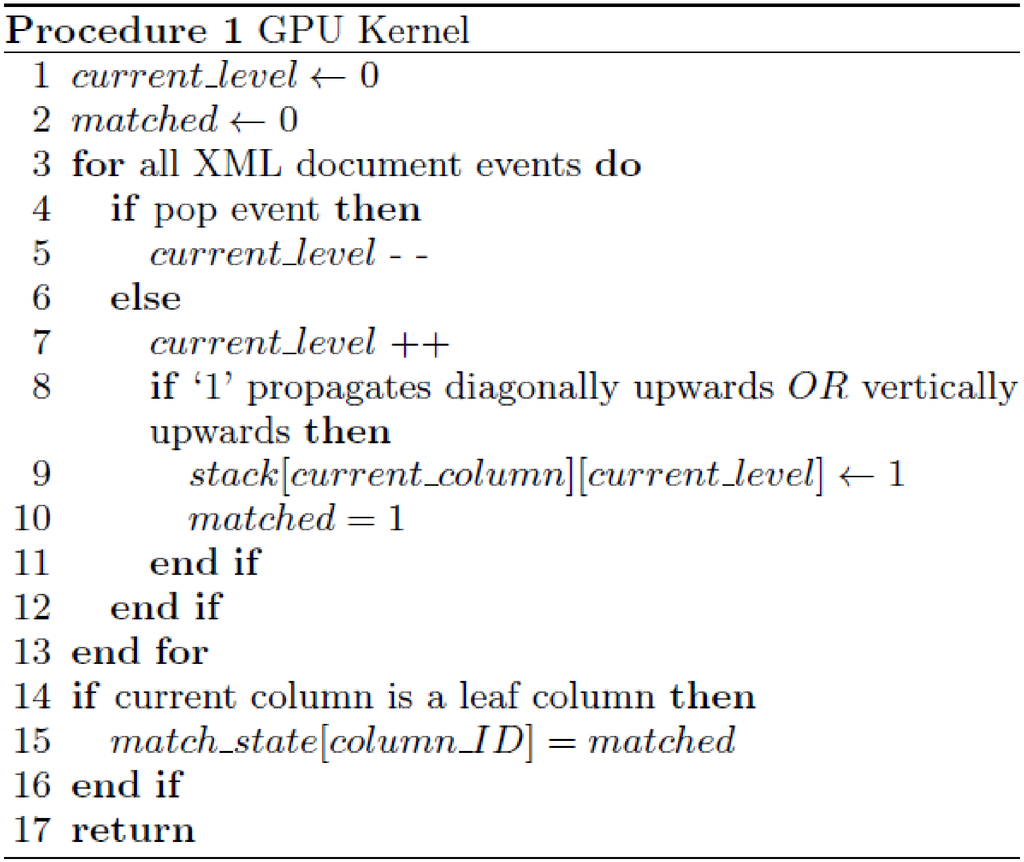


- XML document events are transferred to the GPU, where each entry is 8-bit wide.



- XML document events are stored in **global memory** on the device.

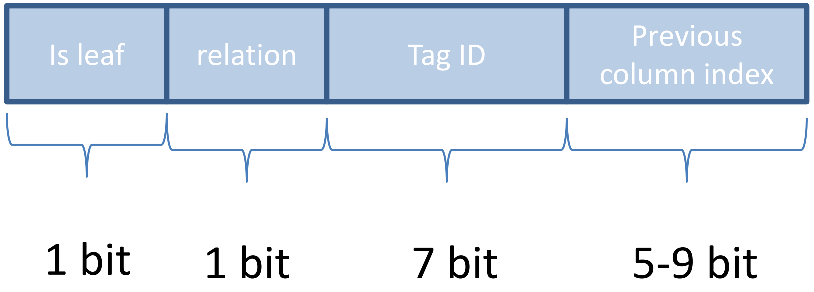
GPU Kernel



Procedure 1 GPU Kernel

```
current_level ← 0
matched ← 0
for all XML document events do
    if pop event then
        current_level - -
    else
        current_level ++
        if '1' propagates diagonally upwards OR vertically upwards then
            stack[current_column][current_level] ← 1
            matched = 1
        end if
    end if
end for
if current column is a leaf column then
    match_state[column_ID] = matched
end if
return
```

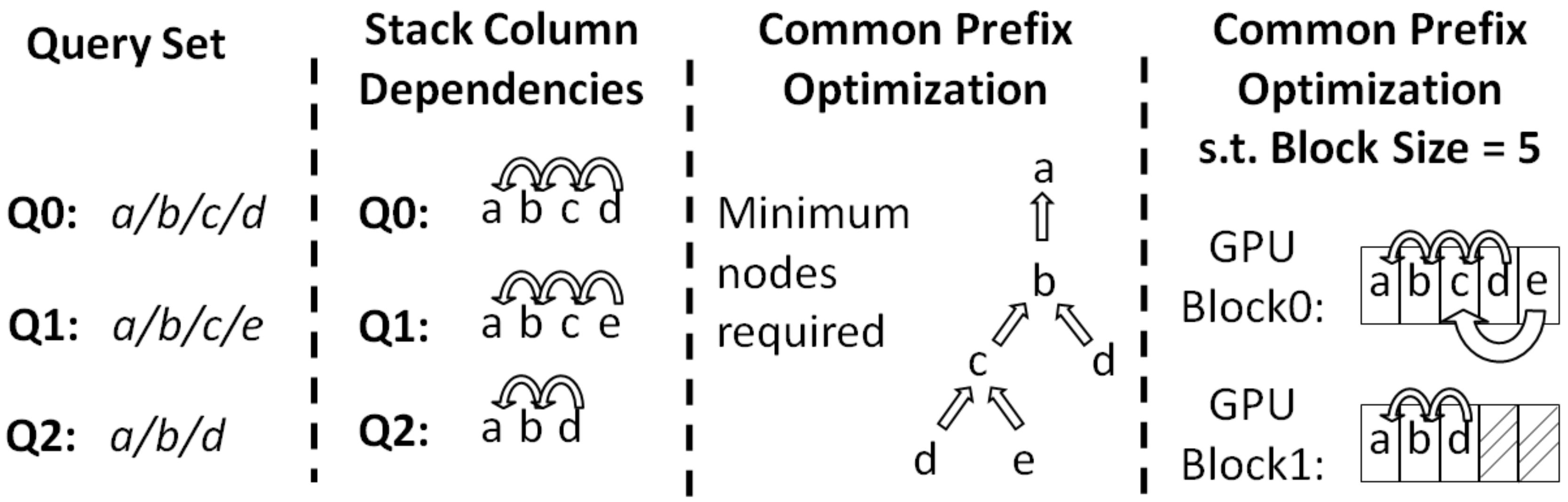
- Each GPU kernel is mapped to a SP (handling the matching of 1 node column).
- Kernels within the same block have a shared memory, thus the node columns belonging to the same query must be in a single block.
- Kernels have personalities, that are initially stored in global memory, then fetched (once) into local variables:



Is leaf	relation	Tag ID	Previous column index
1 bit	1 bit	7 bit	5-9 bit

Common Prefix Optimization

- **Reduce the overall total number of query columns required for computation, by taking advantage of common prefixes across queries**.
  - /a/b/c is common to Q_0 & Q_1
  - /a/b is common to Q_0, Q_1, & Q_2
  - Without optimization, 11 columns (nodes) are required
  - With prefix optimization, 6 columns (nodes) are required
  - With prefix optimization & a physical constraint of 5 on block size, 8 columns (nodes) are required (10 counting empty nodes).



Experimental Evaluation

Experimental Setup

- GPU tests were ran on an NVIDIA TESLA C1060 GPU:
  - 30 SMs comprising of 8 SPs each.
- Software tests comprised of:
  - YFilter software (http://yfilter.cs.umass.edu/code_release.htm).
  - A 2.33GHz Intel Xeon machine with 8GB of RAM running Linux Redhat.

Datasets

DataSets: DBLP, Swissprot, Treebank, and XMark

- **XML Documents**

  - Generated XML documents using ToXgene.

  - Maximum XML tree depth = 16.

  - XML size varied from 5MB – 50MB.

- **Queries**

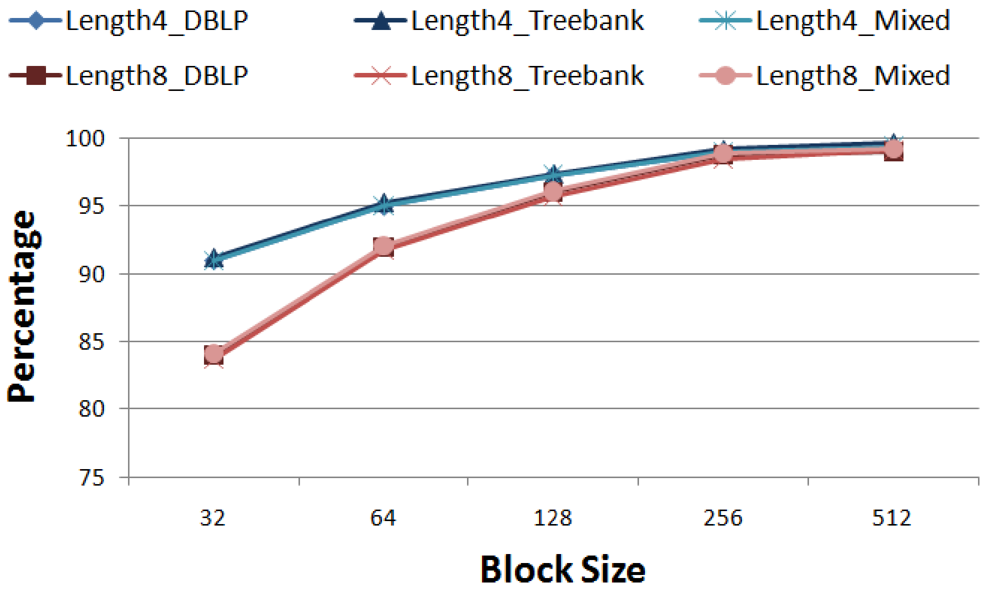
  - Generated twig queries (32 – 128K) using the Yfilter query generator.

  - Maximum path length varied from 4 – 8.

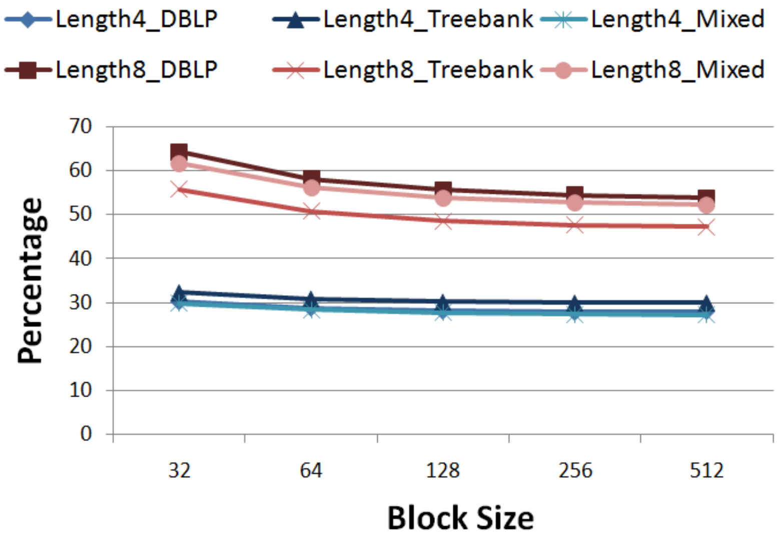
  - Percent occurrence of '//' & '*' varied from 5% - 20%.

Common Prefix Optimization Implementation Evaluation

- Implementation details available in the paper.
- Comparison here versus the **minimal node tree**.
- Varying the block size will affect:
  - Fragmentation (empty nodes).
  - Common prefix size.
  - Repeated nodes across blocks.



Evaluation of the Common Prefix Optimization

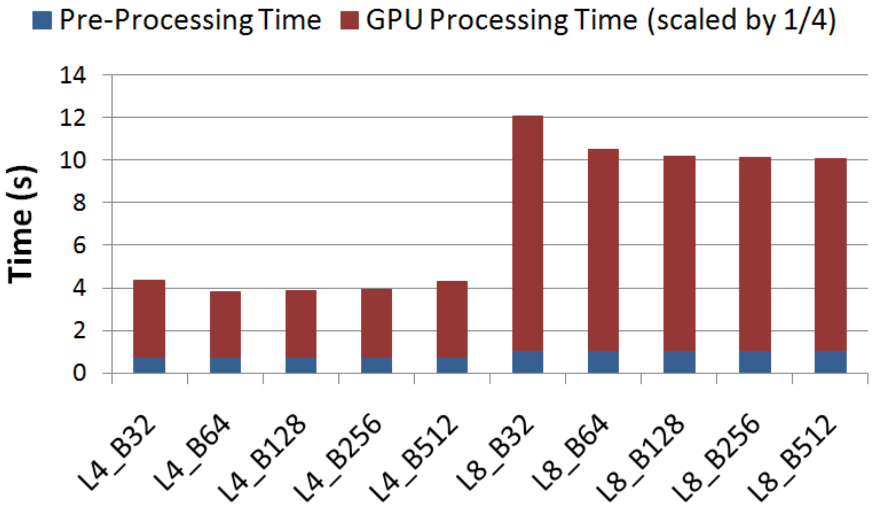


Percentage of Remaining Nodes

- The percentage of remaining nodes includes number of empty nodes.
- Applying the optimization results in:
  - 71% mean reduction of number of nodes on queries of length 4.
  - 45% mean reduction of number of nodes on queries of length 8.

Block Size Effect on the Running Time

- Measured the total processing time with regards to 32K queries for a 50MB XML Document.

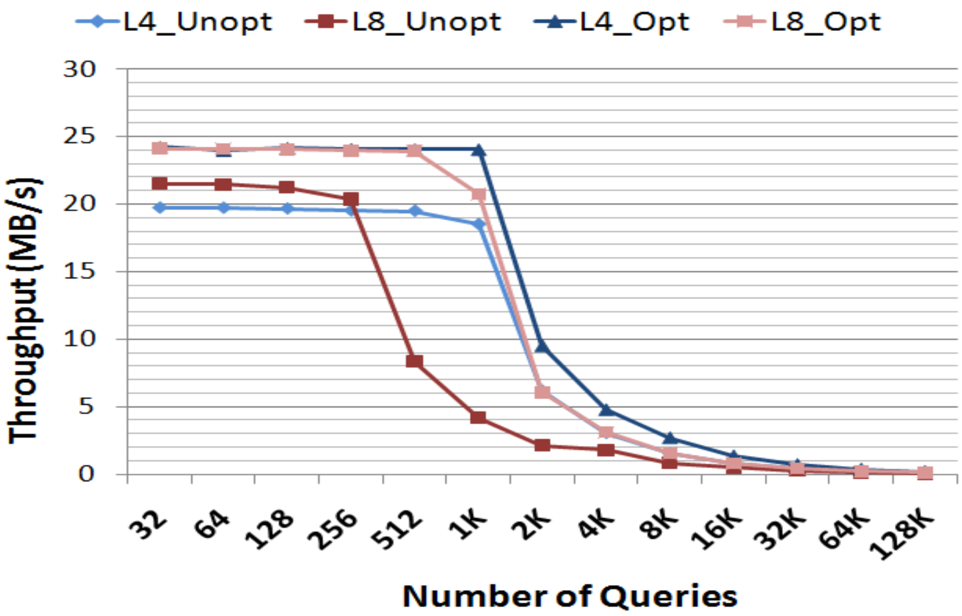


- Tradeoff on block size:
  - Larger block sizes result in higher processing time due to the elevated resource contention.
  - Smaller block sizes result in the least optimized query sets, thus the high processing time.

- Block sizes 128 & 256 result with the least processing time.

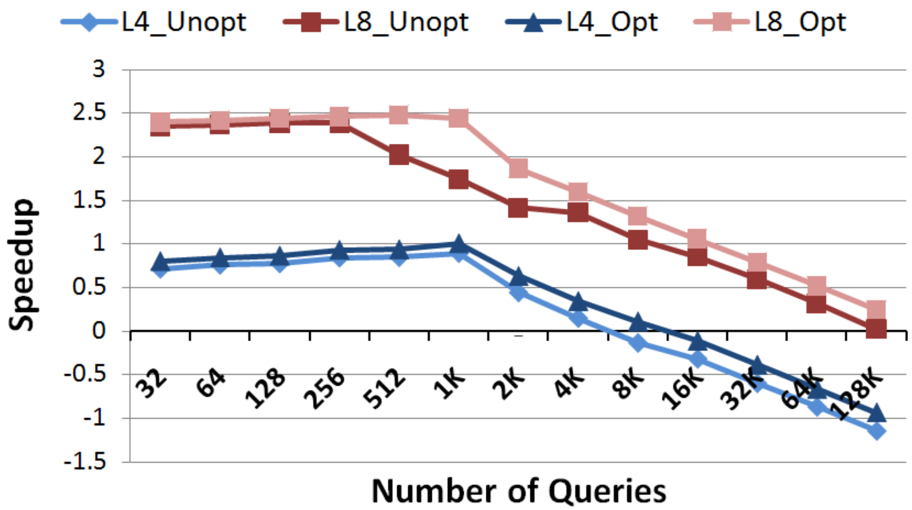
Performance Evaluation

- Measured GPU throughput (MB/s) for filtering algorithm for a 50MB document while varying query dataset size.
- Compared throughput for optimized (Opt) versus un-optimized (Unopt) query configurations.
- Block size is fixed at 128.
- Optimizations increase throughput by a factor of 1.6x.



Speedup vs Software

- Throughput of the proposed GPU-approach is independent of the complexity of XML documents (e.g. recursion) or twig patterns (% occurrence of '//' and '*' in twigs).
- Throughput of software approaches degrades with higher occurrence of '//' and '*' in twig queries.
- Resulting throughput is highest and constant until the GPU is over-utilized.
- Queries of length 8 achieves 2.5 orders of magnitude speedup:
  - Up to 300x **speedup** with an average of 4x.
- Queries of length 4 achieves up to 10x speedup (with an average of 2x), with **slowdown** beyond 16K queries.



Conclusions

- Presented an XML-based path matching system using GPUs.
- The GPU-based framework results in matching of user profiles at a high throughput with minimal update time of query profiles.
- Tens of thousands of profiles can be matched on a GPU (vs thousands on an FPGA).
- Experimental evaluation shows:
  - up to 2.5 order of magnitude speedup (300x) for linear path queries of length 8
  - and up to 10x speedup for length 4 queries, with slowdown beyond 16K queries.

Thank you.

Acknowledgment : NSF CCR 0905509, 0811416, &
NSF IIS 0705916, 0803410, 0910859