GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space

Xiaodong Yu and Michela Becchi
University of Missouri - Columbia

Presented by Zhe Fu
December 11, 2013
Authors

- Xiaodong Yu
  - Addressing out-of-order packets in Deep Packet Inspection (DPI)
    11/2012-Present
    Advisor: Michela Becchi
  - Accelerating the SCOP-fold protein retrieval and classification based on flexible SSE alignment using graphics processing units
    09/2012-present
    Advisor: Michela Becchi
  - Exploring Different Automata Representations for Efficient Regular Expression Matching on GPUs
    09/2011-08/2012
    Advisor: Michela Becchi

- Michela Becchi
  Assistant Professor at Missouri
  M.S. and Ph.D. Degrees at Washington University in St. Louis
  - A-DFA
  - Hybrid FA
  - Implementation on FPGA and NPU
  - Open-source regex tools
  - … …
Regular Expression Matching

- Important in a variety of applications
  - Bibliographic search
  - Protein sequence analysis

- Deep packet inspection
  - search the packet payload against a set of patterns
  - every pattern represents a signature of malicious traffic
  - regular expressions are widely adopted
  - datasets increased in both size and complexity
    - by Dec. 2011, over eleven thousand rules from Snort contain regex
NFA and DFA

- **NFA**
  - limited size
  - expensive per-character processing

- **DFA**
  - limited per character processing
  - large automaton

**A trade-off**

- size of the automaton
- worst-case bound on the amount of per-character processing
NFA and DFA

- a+bc, bcd+ and cde
- input text aabc
Implementation categories

memory-based
- FA stored in memory
- Various parallel platforms:
  - multi-core processors
  - network processors
  - GPUs
- Min memory size and memory bandwidth

logic-based
- FA stored in logic
- Typically target FPGAs
  - updates require platform reprogrammed
- Min logic utilization while allowing fast operation
DFA & Memory based solutions

• Compression mechanisms
  □ Aimed at minimizing the DFA memory footprint
  □ Alphabet reduction, default transition compression, delta-FAs

• Novel automata
  □ Alternative to DFAs
  □ Multiple-DFAs, hybrid-FAs, history-based-Fas, XFAs
GPUs implementation

- Widely used to accelerate a variety of scientific applications
  - Matlab, Bitcoin, …
- Most targeted NVIDIA GPUs
  - CUDA
- Main architectural traits:
  - Streaming Multiprocessors (SMs)
  - Memory hierarchy
  - … …
Recent work

- Most use the coarse-grained block level parallelism
  - support packet- (or flow-) level parallelism intrinsic in networking applications.

- Gnort
  - portion of the dataset compiled into DFA -> GPU
  - rest -> NFA -> CPU
  - DFA memory uncompressed
  - parallelism only at the packet level
    - not leverage any kind of data structure parallelism to further speed up the operation
Recent work

- **XFA -> GPU (31-96)**
  - G80 GPU achieve a 10-11X speedup over Pentium 4
  - Cannot be directly applied to $[^c1..ck]^*$

- **iNFAnt (120 to 543)**
  - First solution applied to rule-sets of arbitrary size and complexity
  - Unpredictable performance & poor worst-case behavior
Recent work

- a GPU design by Zu et al
  - aiming to overcome the limitations of iNFAnt

- Main idea:
  - cluster states into compatibility groups
  - within the same compatibility group cannot be active at the same time

- Drawbacks:
  - requires the exploration of all possible NFA activations
  - equivalent to NFA to DFA transformation
  - limited to datasets consisting of 16-36 regular expressions
  - comparison with iNFAnt is unfair
This paper’s work

- Evaluate GPU designs on **practical datasets**
  - not to show optimal speedup of a given solution on a specific kind of rule-set
  - provide a comprehensive evaluation of automata representations
  - help users to make an informed selection among the plethora of existing algorithmic and data structure proposals
• What is GPGPU?
   - General-Purpose computing on a Graphics Processing Unit
   - Using graphic hardware for non-graphic computations

• What is CUDA?
   - Compute Unified Device Architecture
   - Software architecture for managing data-parallel programming
CPU vs. GPU

- **CPU**
  - Fast caches
  - Branching adaptability
  - High performance

- **GPU**
  - Multiple ALUs (Arithmetic and Logic Unit)
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment/vertex

- CPUs are great for *task* parallelism
- GPUs are great for *data* parallelism
CPU vs. GPU

Figure 1  Floating-Point Operations per Second for the CPU and GPU
CPU vs. GPU

Figure 2  Memory Bandwidth for the CPU and GPU

NSLab, RIIT, Tsinghua Univ
GPU

- Kernel
- Grid
- Block
- Thread
GPU

- global memory
- shared memory
- registers
- local memory
- shared memory
- constant memory
- texture memory
void serial_function(...) {
    ...
}
void other_function(int ... ) {
    ...
}
void saxpy_serial(float ... ) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
void main( ) {
    float x;
    saxpy_serial(..);
    ...
}
CUDA C

CUDA C : C with a few keywords

```c
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

```c
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

NSLab, RIIT, Tsinghua Univ
CUDA Summary

- NVIDIA GPUs comprise a set of Streaming Multiprocessors (SMs), each of them containing a set of simple in-order cores.
- The judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieving good performance.
- CUDA exposes to the programmer two degrees of parallelism:
  - fine-grained parallelism within a thread-block
  - coarse-grained parallelism across multiple thread-blocks
GPU implementation

- General Design
- NFA-based Engines
- DFA-based Engines
- Evaluation
General Design (CPU)

- Supports multiple packet-flows
  - Maps each them onto a different thread-block
- FA is transferred from CPU to GPU only once, at the beginning of the execution
  - Control-flow on CPU consists of a main loop
  - **Step1**: NPF packets –one per packet flow - are transferred from CPU to GPU and stored contiguously on the GPU global memory.
  - **Step2**: the FA traversal kernel is invoked, thus triggering the regex matching process on GPU.
  - **Step3**: The result of the matching operation is transferred from GPU to CPU at the end of the flow-traversal.
- Dual buffering to hide the CPU-GPU packet transfer time
NFA-based Engines

• iNFAnt
  - Most efficient and broadly applicable GPU-based NFA proposal
  - Transitions are represented through a list of (source, destination) pairs sorted by their triggering symbol
  - An ancillary data structure records, for each symbol, the first transition within the transition list
  - Persistent states are handled separately using a state vector.
iNFAnt

1: \( \text{current}_{sv} \leftarrow \text{initial}_{sv} \)
2: while \( \neg \text{input.} \text{empty} \) do
3: \( c \leftarrow \text{input.first} \)
4: \( \text{input} \leftarrow \text{input.tail} \)
5: \( \text{future}_{sv} \leftarrow \text{current}_{sv} \cap \text{persistent}_{sv} \)
6: while a transition on \( c \) is pending do
7: \( \text{src} \leftarrow \text{transition source} \)
8: \( \text{dst} \leftarrow \text{transition destination} \)
9: if \( \text{current}_{sv}[\text{src}] \) is set then
10: \( \text{atomicSet}(\text{future}_{sv}, \text{dst}) \)
11: end if
12: end while
13: \( \text{current}_{sv} \leftarrow \text{future}_{sv} \)
14: end while
15: return \( \text{current}_{sv} \)
Inefficiency of iNFAnt

- On large NFAs, transition list can be **very long** and may require a **large number** of thread-block iterations
- In most traversal steps, *only a minority* of the NFA states are active.

**Optimization:**

- NFA states can be easily clustered into groups of states that cannot be active at the same time.

```
1: \texttt{current}_{sv} \leftarrow \texttt{initial}_{sv}
2: \textbf{while} \ \neg \texttt{input}.empty \ \textbf{do}
3: \quad \texttt{c} \leftarrow \texttt{input}.first
4: \quad \texttt{input} \leftarrow \texttt{input}.tail
5: \quad \texttt{future}_{sv} \leftarrow \texttt{current}_{sv} \land \texttt{persistent}_{sv}
6: \textbf{while} \ a \ transition \ on \ c \ is \ pending \ \textbf{do}
7: \quad \texttt{src} \leftarrow \text{transition source}
8: \quad \texttt{dst} \leftarrow \text{transition destination}
9: \quad \textbf{if} \ \texttt{current}_{sv}[\texttt{src}] \ \textbf{is} \ set \ \textbf{then}
10: \quad \quad \texttt{atomicSet}(\texttt{future}_{sv}, \texttt{dst})
11: \quad \quad \textbf{end if}
12: \textbf{end while}
13: \texttt{current}_{sv} \leftarrow \texttt{future}_{sv}
14: \textbf{end while}
15: \textbf{return} \ \texttt{current}_{sv}
```
Optimized NFA

- Optimization 1
  - Transitions on the same character are *sorted* according to the source state identifier.
  - *Largest* active state identifier SIDMAX is *known*.
  - Execution of loop 6 can be *terminated* when the first transition with source identifier *greater* than SIDMAX is encountered.

```plaintext
1: current_{sv} ← initial_{sv}
2: while ¬input.empty do
3:   c ← input.first
4:   input ← input.tail
5:   future_{sv} ← current_{sv} ∧ persistent_{sv}
6:   while a transition on c is pending do
7:     src ← transition source
8:     dst ← transition destination
9:     if current_{sv}[src] is set then
10:        atomicSet(future_{sv}, dst)
11:    end if
12:   end while
13:   current_{sv} ← future_{sv}
14: end while
15: return current_{sv}
```

(a) NFA transition graph.

(b) Transition vector.
Optimized NFA-based design

- Optimization 2
  - States can be grouped according to their incoming transitions
  - Define two or more states as compatible if they can potentially be active at the same time
  - States with a single incoming transition on character $c_i$ are grouped into group $c_i$
  - The other states are grouped into a special group
Optimized NFA-based design

- Optimization 2
  - States belonging to the same group are compatible
  - States belonging to different group \( g_i \) are incompatible
  - States belonging to group \( g_{\text{overlap}} \) are compatible with any other state
Optimized NFA-based design

- Optimization 3
  - combine the benefits of the two optimizations
  - adopt a proper numbering scheme
    - persistent states and states with self-loops belong to the group $\text{overlap}$
    - states close to the NFA entry state are more likely to be traversed
    - benefits of optimization 1 will be higher if the state numbering scheme is such that states within the same compatibility group have similar identifiers.
Optimized NFA-based design

• Optimization 3
• **Step 1:** number the states according to a breadth-first traversal of the NFA
• **Step 2:** compute the compatibility groups
• **Step 3:** order the compatibility groups
  - $\text{group}_{\text{overlap}}$
  - sort all $\text{group}_{ci}$ according to the average frequency of character $c$
Optimized NFA-based design

![Graph showing state transitions](image)

<table>
<thead>
<tr>
<th>Orig. ID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>New ID</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>13</td>
<td>1</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>17</td>
<td>2</td>
<td>18</td>
<td>15</td>
<td>3</td>
<td>8</td>
<td>16</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>Orig. state ID</th>
<th>New state ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2, 12, 16, 17</td>
<td>4, 5, 6, 7</td>
</tr>
<tr>
<td>b</td>
<td>14, 18</td>
<td>8, 9</td>
</tr>
<tr>
<td>c</td>
<td>1, 5, 6</td>
<td>10, 11, 12</td>
</tr>
<tr>
<td>d</td>
<td>3, 7, 11, 15</td>
<td>13, 14, 15, 16</td>
</tr>
<tr>
<td>e</td>
<td>8, 10, 19</td>
<td>17, 18, 19</td>
</tr>
</tbody>
</table>
### Optimized NFA-based design

#### Before renaming

<table>
<thead>
<tr>
<th>Character</th>
<th>Src. vector</th>
<th>Dest. vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>13</td>
</tr>
</tbody>
</table>

#### After renaming

<table>
<thead>
<tr>
<th>Character</th>
<th>Src. vector</th>
<th>Dest. vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

---

**Group overlap**

**Group vector**

**Character offset vector**

---

NSLab, RIIT, Tsinghua Univ
Optimized NFA-based design

iNFAn't implementation

Two optimizations applied to iNFAn't
DFA-based Engines

- The number of active states can vary from iteration to iteration
- So does the amount of work performed in different phases of the traversal
- DFAs can offer predictable and bounded per-character processing

**But:**
- Repetitions of wildcards and large character sets may lead to state explosion
- .* [^abc] *
DFA-based Engines

- Patterns can be handled by grouping clusters and by generating multiple DFAs
  - Rather than statically determining the number of DFAs to generate, the user defines the maximum allowed DFA size
  - Try to cluster together as many regular expressions as possible
- Set the maximum DFA size to 64K
- Allow the use of 16-bit state identifiers
DFA-based Engines

• Uncompressed DFA-based solution
  - DFAs are laid out next to one another in global memory
  - Current active state in a local register

```c
kernel uncompressed-DFA
1:  currents ← initialsv[tid]
2:  while !input.empty do
3:      c ← input.first
4:      input ← input.tail
5:      currents ← state_table[tid][currents][c];
6:      initialsv[tid] ← currents
end
```
Compressed DFA-based solution

Connect pair of states $S_1$ and $S_2$ characterized by transition commonality through non-consuming directed default transitions.

```plaintext
kernel compressed-DFA
1: current_sv[tid.y] ← initial_sv[tid.y]
2: idx[tid.y] ← 0
3: while (idx[tid.y] != PACKET_SIZE) do
4: future_sv[tid.y] ← INVALID
5: c ← input[idx[tid.y]]
6: tx_offset ← offset[current_sv[tid.y]]
7: while current states has unprocessed transitions
8: symbol ← labeled_tx[tid.y][tx_offset][it_offset+tid.x].char
9: dst ← labeled_tx[tid.y][tx_offset][it_offset+tid.x].dst
10: if (symbol = c)
11: future_sv[tid.y] ← dst
12: idx[tid.y]++
13: if (future_sv[tid.y] = INVALID)
14: future_sv[tid.y] ← default_tx[tid.y][current_sv[tid.y]]
15: current_sv[tid.y] = future_sv[tid.y]
16: initial_sv[tid.y] ← current_sv[tid.y]
end
```
Compressed DFA-based solution

- Default transitions stored in a one dimensional array
- Labeled transitions can be represented through a list of (input character, destination state) pairs, and an ancillary data structure (offset array) indicating
  - Store labeled transitions in a one-dimensional array of 32 bits
  - For each element, 8 bits are used to represent the ASCII input character, and the remaining bits to store the state identifier (up to 16M states)
  - Stored in *global memory*
  - Frequently accessed initial state in *constant memory*
Compressed DFA-based solution

- Parallelism in **two** ways:
  - Different threads process different DFAs
  - Within the same DFA, different labeled transitions

- Accomplished by using bidimensional thread-blocks

```c
kernel compressed-DFA
1: currentsv[tid.y] = initialsv[tid.y]
2: idx[tid.y] = 0
3: while (idx[tid.y] != PACKET_SIZE) do
4:     futuresv[tid.y] = INVALID
5:     c = input[idx[tid.y]]
6:     tx_offset = offset[currentsv[tid.y]]
7:     while current states has unprocessed transitions
8:         symbol = labeled_tx[tid.y][tx_offset][it_offset+tid.x].char
9:         dst = labeled_tx[tid.y][tx_offset][it_offset+tid.x].dst
10:        if (symbol = c)
11:            futuresv[tid.y] = dst
12:            idx[tid.y]++
13:        if (futuresv[tid.y] = INVALID)
14:            futuresv[tid.y] = default_tx[tid.y][currentsv[tid.y]]
15:        currentsv[tid.y] = futuresv[tid.y]
16:        initialsv[tid.y] = currentsv[tid.y]
end
```
After default transition compression, more than 90% of the states are left with 4-5 labeled transitions.
Enhanced compressed DFA-based solution

- Allow compressed states to consist of either 4 or 8 labeled transitions
- All states with more than 8 transitions are represented in full and processed via direct indexing
EXPERIMENTAL EVALUATION

- Data Sets and Platform
  - Intel Xeon E5620 CPU
  - an NVIDIA GTX 480 GPU
    - $250
    - 15 streaming multiprocessor
    - each consisting of 32 cores
    - 1.5 GB of global memory
    - CentOS 5.5 and CUDA 4

<table>
<thead>
<tr>
<th>Dataset</th>
<th># regex</th>
<th>NFA</th>
<th>DFA</th>
<th>Uncompressed-DFA</th>
<th>Compressed-DFA</th>
<th>Enhanced-DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># states</td>
<td># tx</td>
<td>Mem (MB)</td>
<td># states</td>
<td>Mem (MB)</td>
</tr>
<tr>
<td>Backdoor</td>
<td>226</td>
<td>4.3k</td>
<td>70.8k</td>
<td>0.54</td>
<td>13</td>
<td>960.1k</td>
</tr>
<tr>
<td>Spyware</td>
<td>462</td>
<td>7.7k</td>
<td>66.8k</td>
<td>0.51</td>
<td>19</td>
<td>680.2k</td>
</tr>
<tr>
<td>EM</td>
<td>1k</td>
<td>28.7k</td>
<td>51.9k</td>
<td>0.40</td>
<td>1</td>
<td>28.7k</td>
</tr>
<tr>
<td>Range.5</td>
<td>1k</td>
<td>28.5k</td>
<td>91.8k</td>
<td>0.70</td>
<td>1</td>
<td>41.8k</td>
</tr>
<tr>
<td>Range.1</td>
<td>1k</td>
<td>29.6k</td>
<td>117.9k</td>
<td>0.90</td>
<td>1</td>
<td>54.4k</td>
</tr>
<tr>
<td>Dotstar.05</td>
<td>1k</td>
<td>29.1k</td>
<td>116.8k</td>
<td>0.89</td>
<td>13</td>
<td>251k</td>
</tr>
<tr>
<td>Dotstar.1</td>
<td>1k</td>
<td>29.2k</td>
<td>115.7k</td>
<td>0.88</td>
<td>21</td>
<td>603.8k</td>
</tr>
<tr>
<td>Dotstar.2</td>
<td>1k</td>
<td>28.7k</td>
<td>114.6k</td>
<td>0.87</td>
<td>32</td>
<td>1.6M</td>
</tr>
</tbody>
</table>
### Performance evaluation

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CPU</th>
<th>GPU</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NFA</td>
<td>DFA</td>
<td>NFA</td>
<td>DFA</td>
</tr>
<tr>
<td>Backdoor</td>
<td>0.4</td>
<td>7.9</td>
<td>40.8</td>
<td>37.6</td>
</tr>
<tr>
<td>Spyware</td>
<td>0.9</td>
<td>4.0</td>
<td>40.1</td>
<td>46.4</td>
</tr>
<tr>
<td>E-M</td>
<td>3.9</td>
<td>40.3</td>
<td>14.3</td>
<td>27.3</td>
</tr>
<tr>
<td>Range,5</td>
<td>4.9</td>
<td>40.6</td>
<td>13.6</td>
<td>26.7</td>
</tr>
<tr>
<td>Range1</td>
<td>4.9</td>
<td>41.1</td>
<td>18.6</td>
<td>25.7</td>
</tr>
<tr>
<td>Dotstar,05%</td>
<td>1.4</td>
<td>7.9</td>
<td>20.0</td>
<td>25.8</td>
</tr>
<tr>
<td>Dotstar,1</td>
<td>0.9</td>
<td>5.2</td>
<td>18.7</td>
<td>26.0</td>
</tr>
<tr>
<td>Dotstar,2</td>
<td>0.6</td>
<td>3.2</td>
<td>18.6</td>
<td>24.0</td>
</tr>
<tr>
<td>Backdoor</td>
<td>0.4</td>
<td>7.4</td>
<td>37.2</td>
<td>31.9</td>
</tr>
<tr>
<td>Spyware</td>
<td>0.4</td>
<td>4.3</td>
<td>35.3</td>
<td>35.9</td>
</tr>
<tr>
<td>E-M</td>
<td>3.1</td>
<td>42.4</td>
<td>10.0</td>
<td>24.9</td>
</tr>
<tr>
<td>Range,5</td>
<td>3.3</td>
<td>42.3</td>
<td>16.9</td>
<td>24.2</td>
</tr>
<tr>
<td>Range1</td>
<td>3.3</td>
<td>42.0</td>
<td>13.3</td>
<td>23.3</td>
</tr>
<tr>
<td>Dotstar,05%</td>
<td>0.5</td>
<td>7.6</td>
<td>13.2</td>
<td>17.6</td>
</tr>
<tr>
<td>Dotstar,1</td>
<td>0.6</td>
<td>5.2</td>
<td>16.4</td>
<td>20.8</td>
</tr>
<tr>
<td>Dotstar,2</td>
<td>0.8</td>
<td>3.0</td>
<td>14.6</td>
<td>19.4</td>
</tr>
</tbody>
</table>

**Note:** The table above illustrates performance evaluation results for various datasets using CPU and GPU. The performance metrics include NFA, DFA, O-NFA, U-DFA, C-DFA, and E-DFA. The performance factor $P_M$ is varied from 0.35 to 0.95.
Performance evaluation

Table 3: Effect of number of flows/SM on performance.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Optimal # flows per SM</th>
<th>Improvement over 1 flow per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
</tr>
<tr>
<td>U-DFA</td>
<td>5</td>
<td>1.72</td>
</tr>
<tr>
<td>C-DFA</td>
<td>5 (single-DFA)</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>2-3 (multi-DFAs)</td>
<td></td>
</tr>
<tr>
<td>E-DFA</td>
<td>5</td>
<td>2.49</td>
</tr>
<tr>
<td>iNFAnt</td>
<td>4</td>
<td>1.81</td>
</tr>
<tr>
<td>Opt-iNFAnt</td>
<td>4</td>
<td>2.55</td>
</tr>
</tbody>
</table>

Table 4: Effect of caching: % miss rate (MR) and performance improvement (PI) on different datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>$P_M=0.35$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MR</td>
<td>PI</td>
<td>MR</td>
<td>PI</td>
<td>MR</td>
<td>PI</td>
<td>MR</td>
<td>PI</td>
</tr>
<tr>
<td>Backdoor</td>
<td>0.34</td>
<td>1.65</td>
<td>1.30</td>
<td>1.67</td>
<td>7.69</td>
<td>1.59</td>
<td>7.00</td>
<td>1.60</td>
</tr>
<tr>
<td>Spyware</td>
<td>0.59</td>
<td>1.52</td>
<td>3.47</td>
<td>1.53</td>
<td>8.06</td>
<td>1.41</td>
<td>17.91</td>
<td>1.36</td>
</tr>
<tr>
<td>Dotstar.05</td>
<td>2.77</td>
<td>1.66</td>
<td>8.35</td>
<td>1.54</td>
<td>35.14</td>
<td>1.09</td>
<td>37.44</td>
<td>0.98</td>
</tr>
<tr>
<td>Dotstar.1</td>
<td>2.81</td>
<td>1.37</td>
<td>7.49</td>
<td>1.40</td>
<td>15.40</td>
<td>1.25</td>
<td>36.05</td>
<td>1.16</td>
</tr>
<tr>
<td>Dotstar.2</td>
<td>6.07</td>
<td>1.16</td>
<td>4.44</td>
<td>1.15</td>
<td>10.16</td>
<td>1.10</td>
<td>39.57</td>
<td>0.93</td>
</tr>
</tbody>
</table>
Conclusion

• A comprehensive study of regular expression matching on GPUs
  - Datasets of practical size and complexity
  - Explored advantages and limitations of different NFA- and DFA-based representations

• Uncompressed DFA solution outperforms other implementations
  - On large and complex datasets exceeding the memory capacity
  - Schemes to improve a basic default-transition compressed DFA design
Thanks!