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

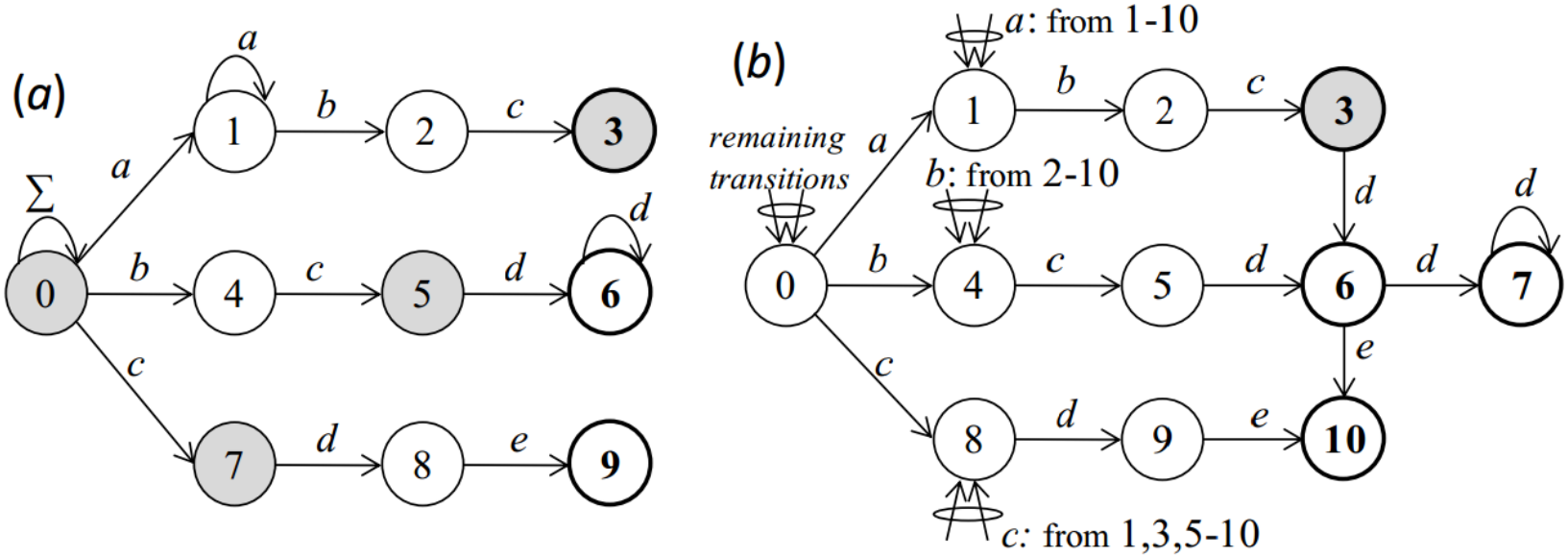
NFA and DFA

- 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

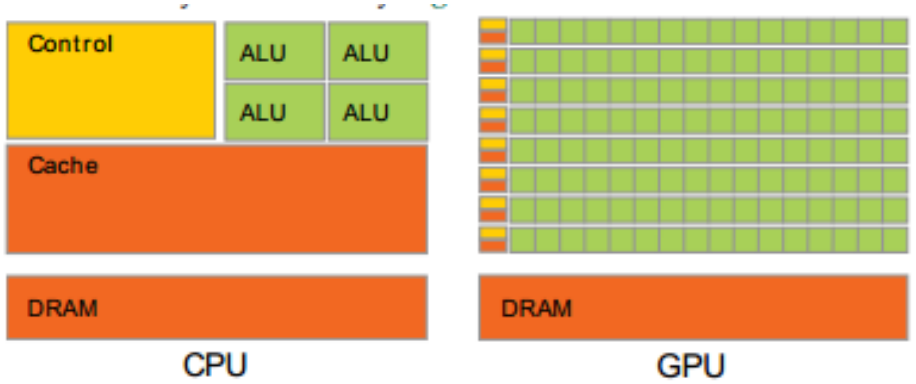
GPU implementation

- What is **GPGPU**?
  - **G**eneral-**P**urpose computing on a **G**raphics **P**rocessing **U**nit
  - Using graphic hardware for non-graphic computations

- What is **CUDA**?
  - **C**ompute **U**nified **D**evice **A**rchitecture
  - 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

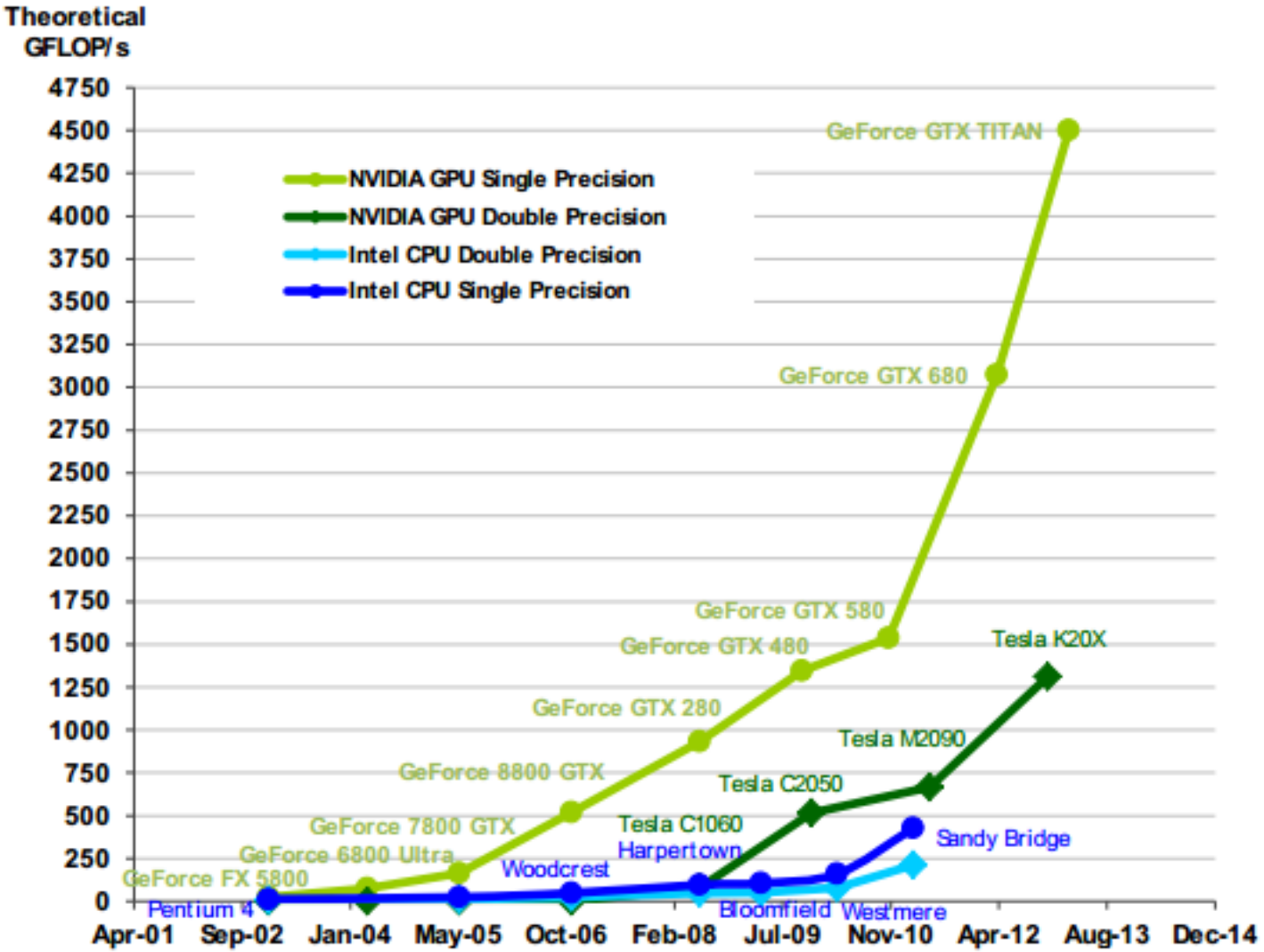


Figure 1 Floating-Point Operations per Second for the CPU and GPU

CPU vs. GPU

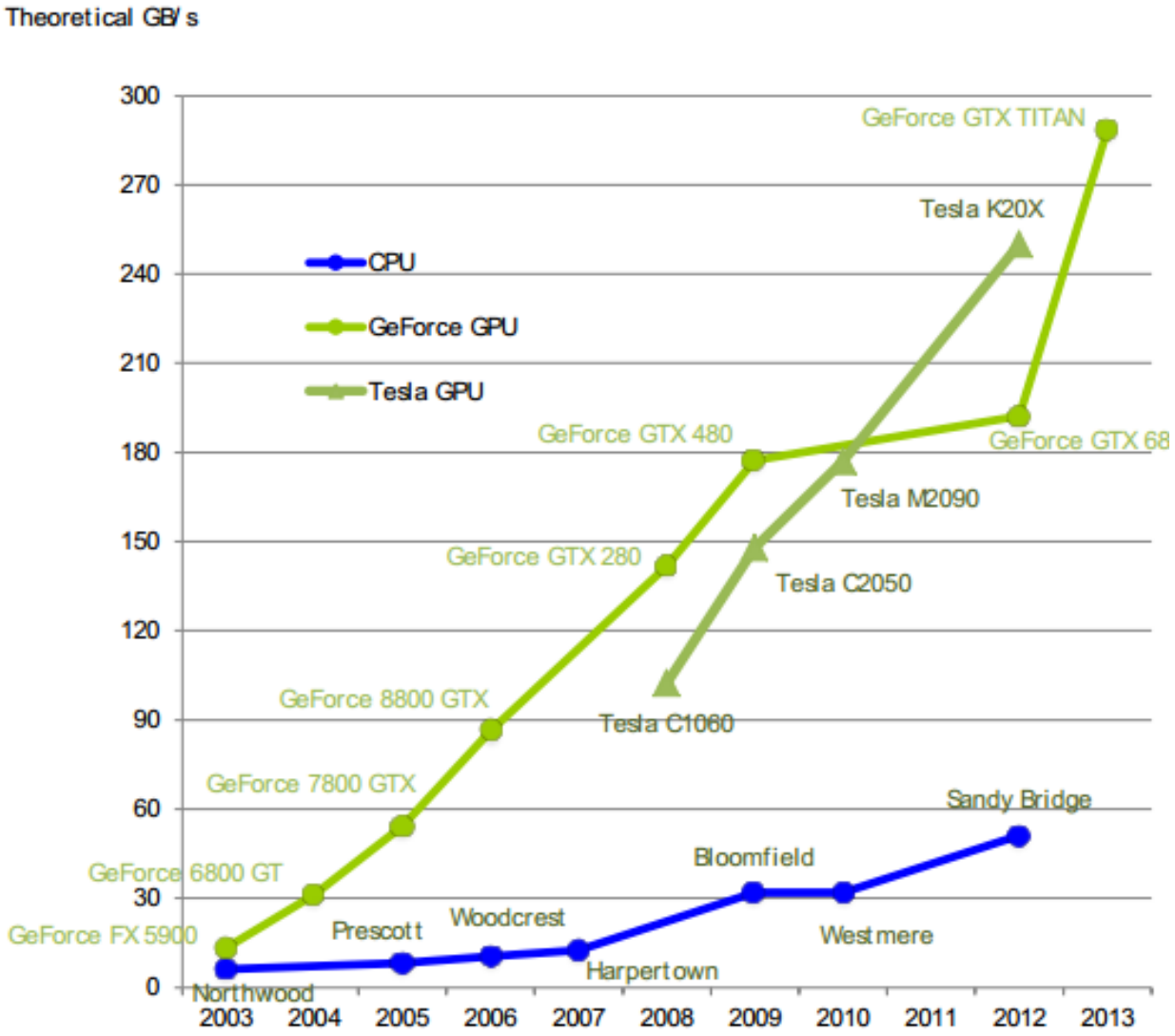
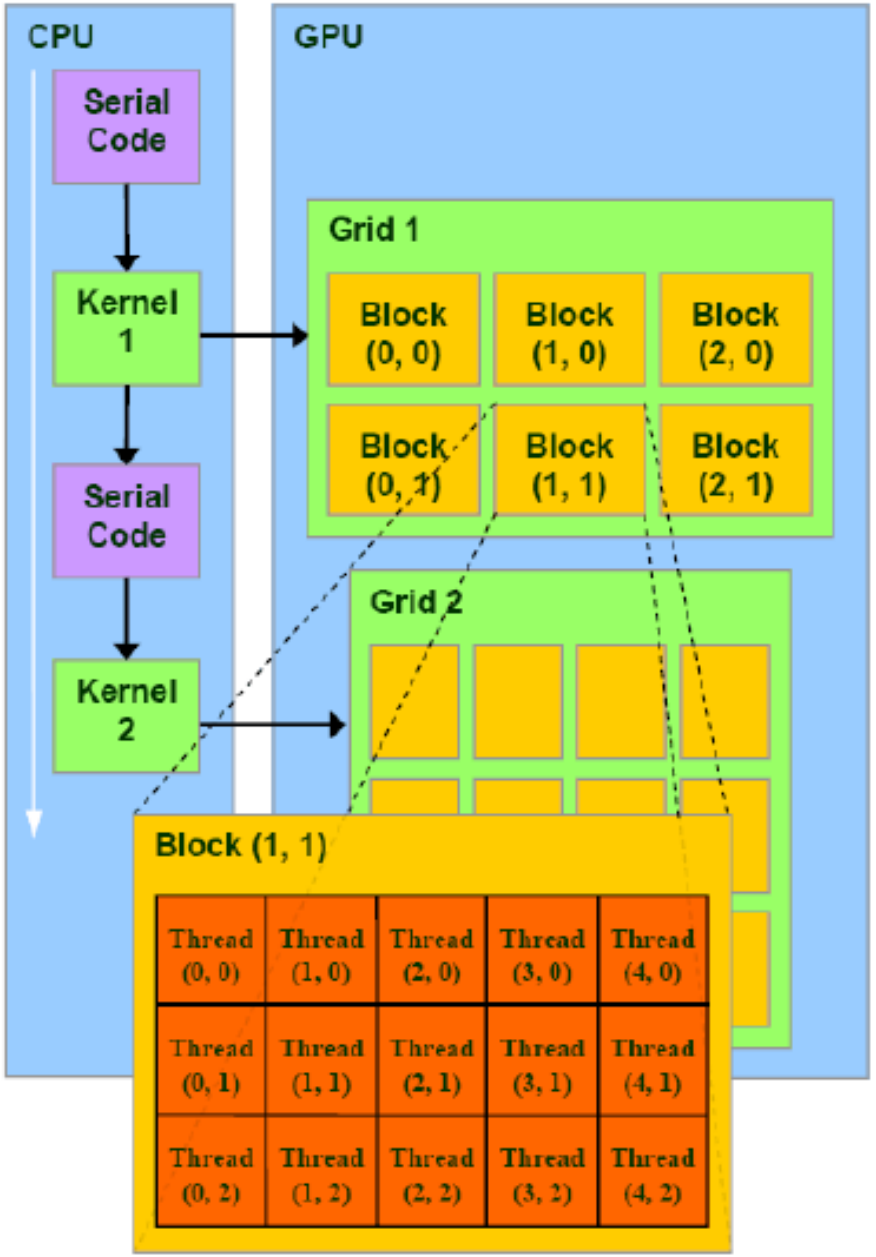


Figure 2 Memory Bandwidth for the CPU and GPU

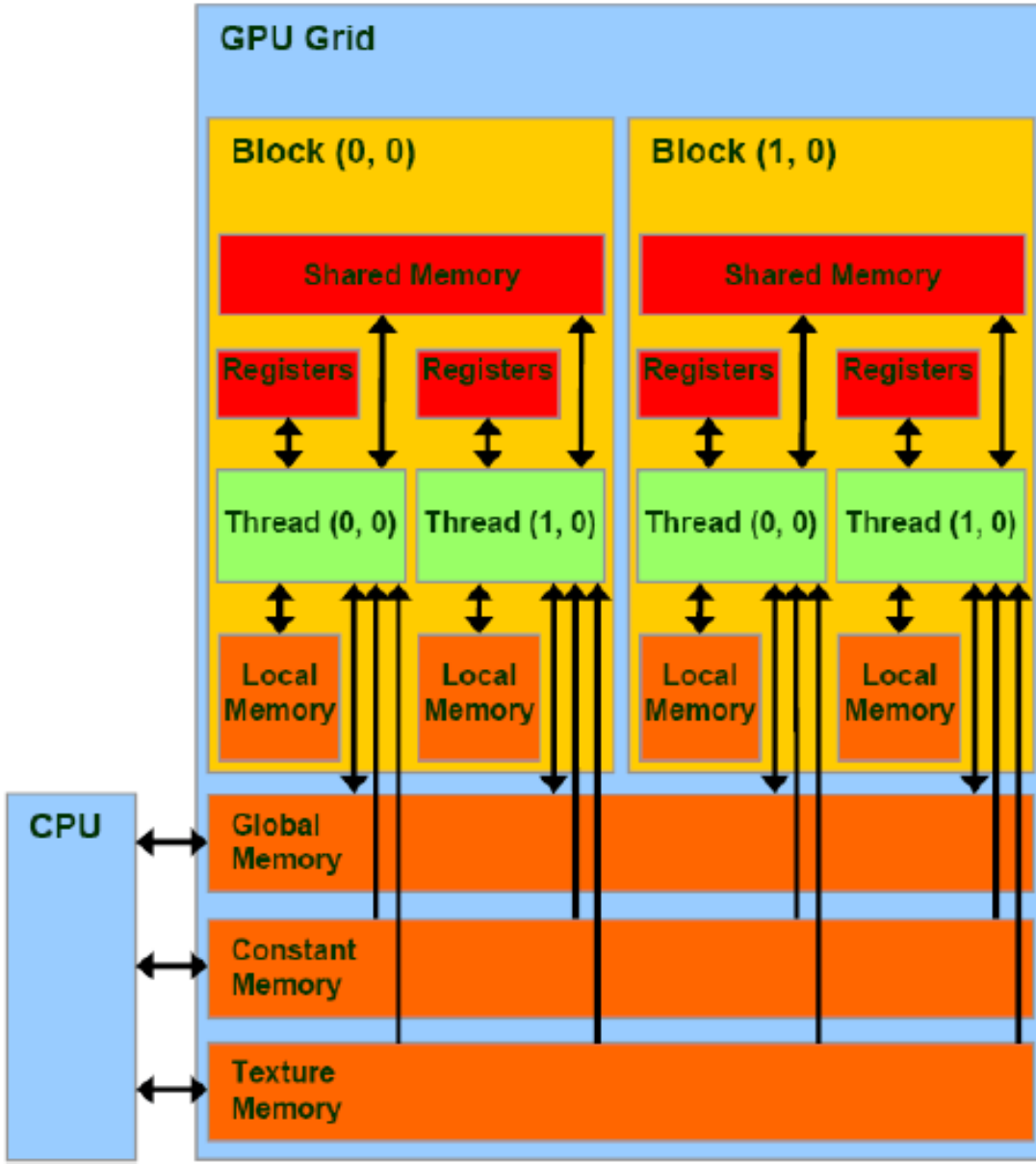
GPU

- Kernel
- Grid
- Block
- Thread

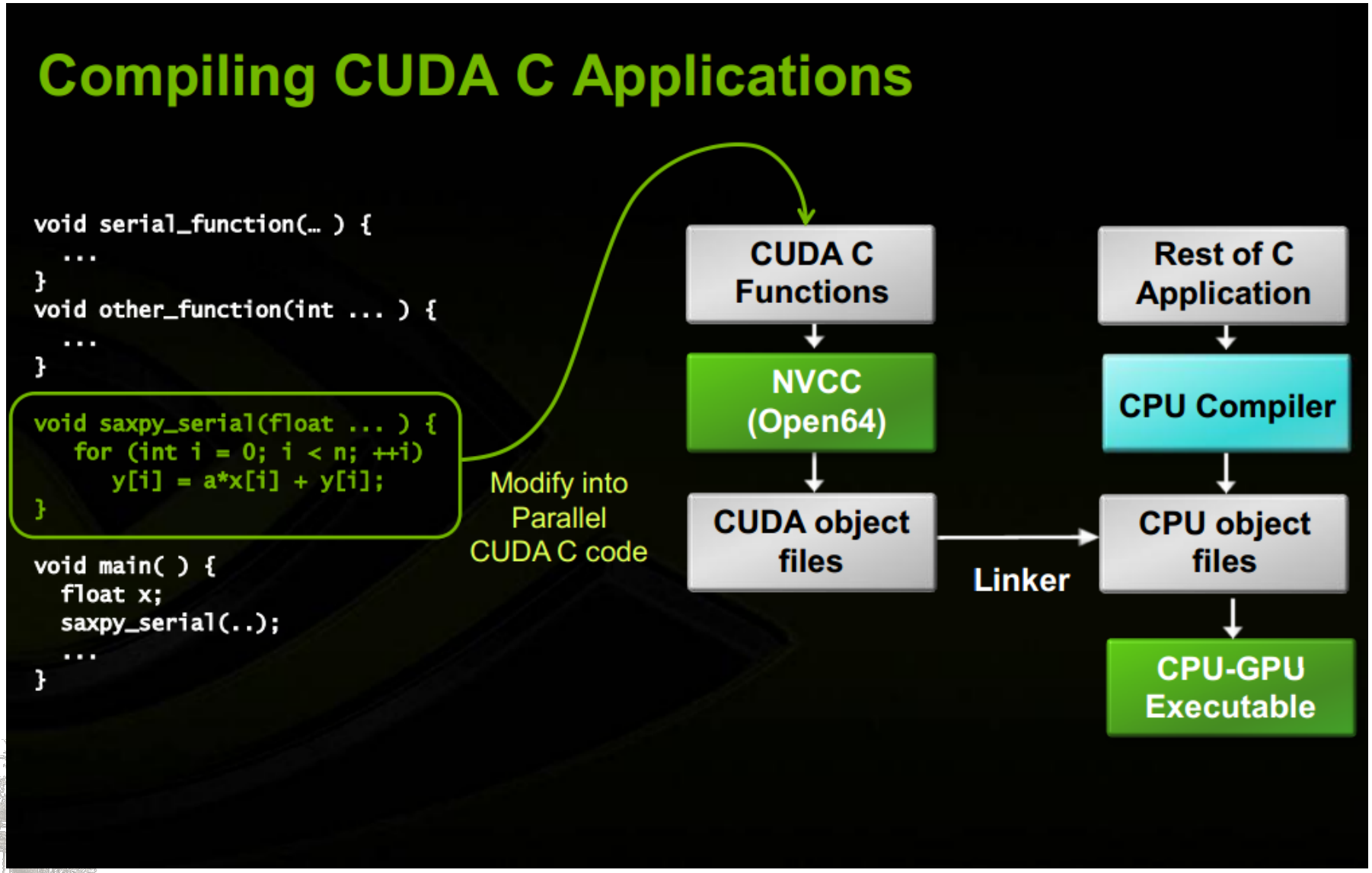


GPU

- global memory
- shared memory
- registers
- local memory
- shared memory
- constant memory
- texture memory



Compiling CUDA C



CUDA C

CUDA C : C with a few keywords

```
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);
```

Standard C Code

```
__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);
```

Parallel C Code

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

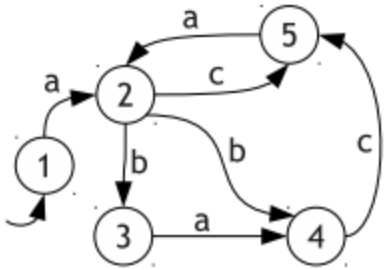
General Design (CPU)

- 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

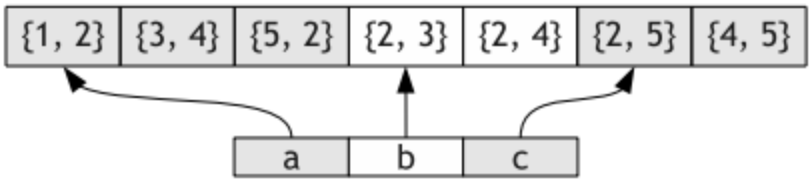
NFA-based Engines

- 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



(a) NFA transition graph.



(b) Transition vector.

```
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
```

Inefficiency of iNFAnt

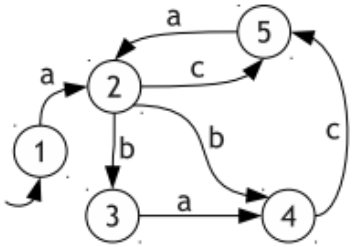
- On large NFAs, transition list can be ***very long*** and may require a ***large number*** of thread-block iterations
- In most traversal steps, ***only a minority*** of the NFA states are active.
- Optimization:
  - NFA states can be easily clustered into groups of states that cannot be active at the same time.

```
1:  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
```

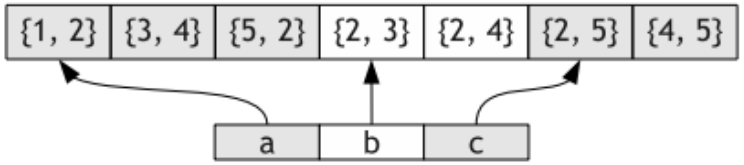
Optimized NFA

- 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

```
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 ci are grouped into group ci
  - 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 $group_{ci}$ are incompatible
  - States belonging to $group_{overlap}$ are compatible with any other state

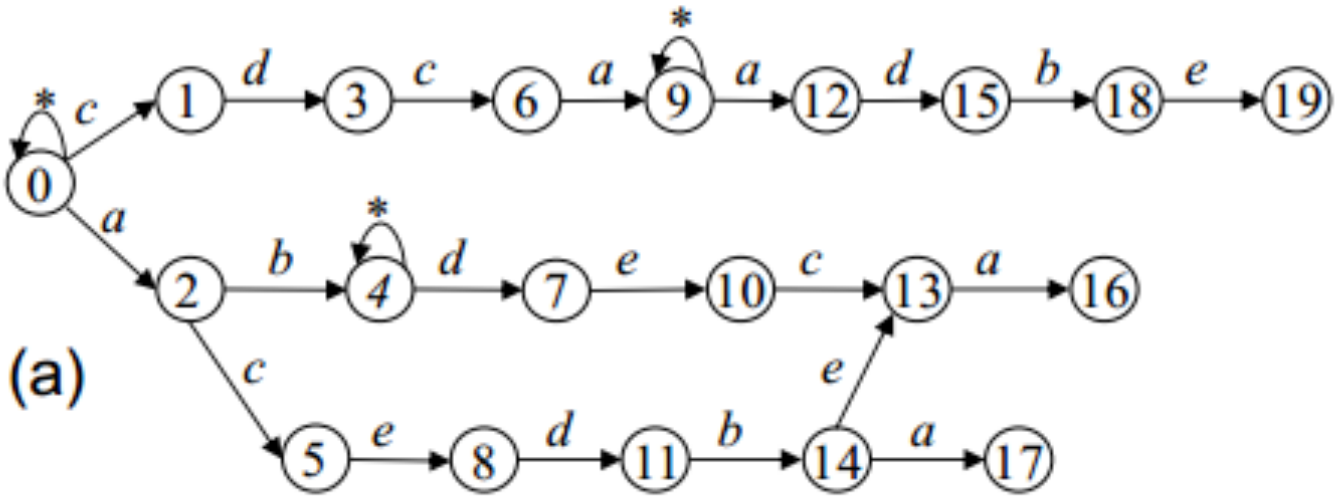
Optimized NFA-based design

- Optimization 3
  - combine the benefits of the two optimizations
  - adopt a proper numbering scheme
    - persistent states and states with self-loops belong to the $group_{overlap}$
    - 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
- **Step1:** number the states according to a breadth-first traversal of the NFA
- **Step2:** compute the compatibility groups
- **Step3:** order the compatibility groups
  - $group_{overlap}$
  - sort all $group_{ci}$ according to the average frequency of character c

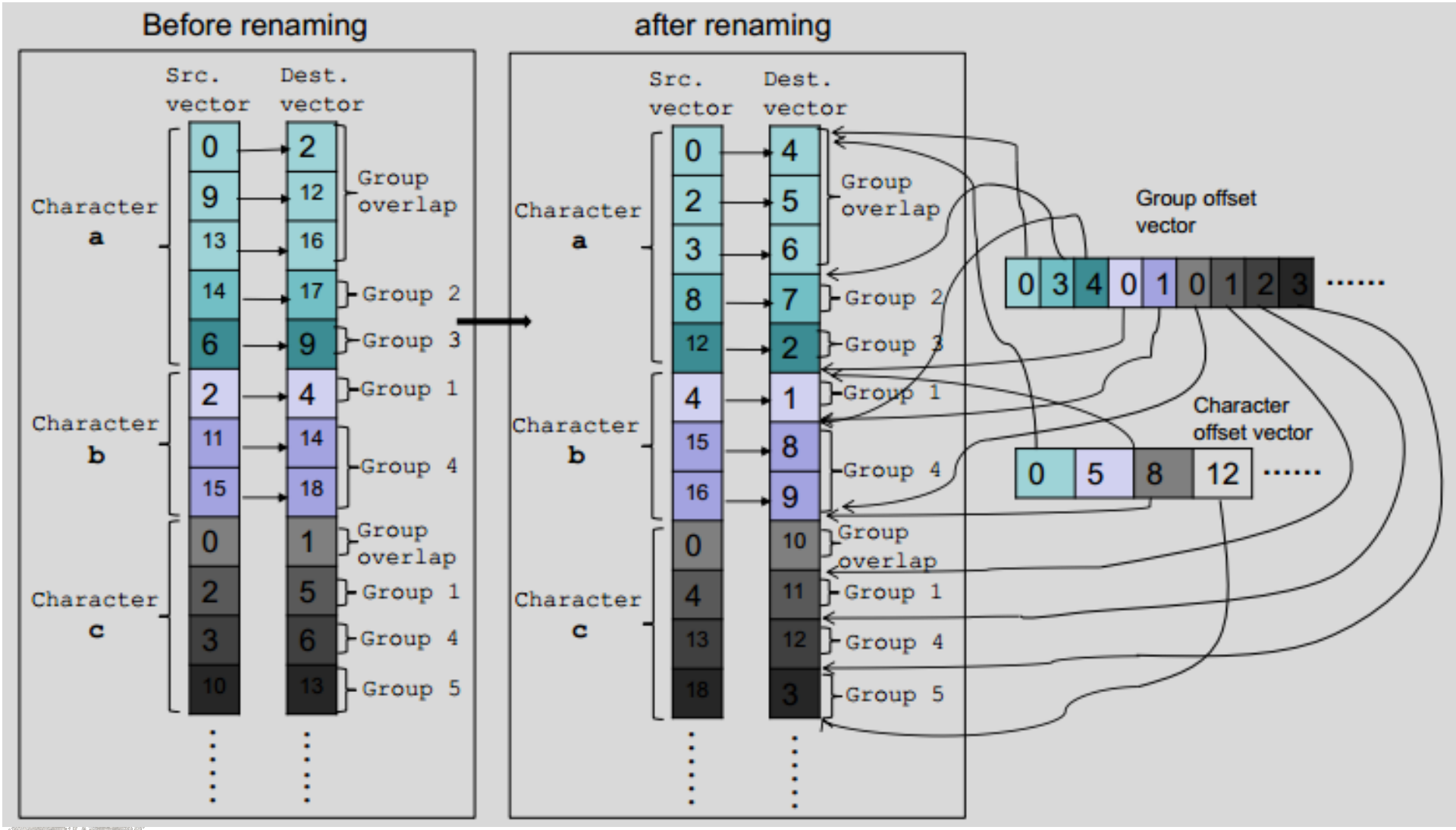
Optimized NFA-based design



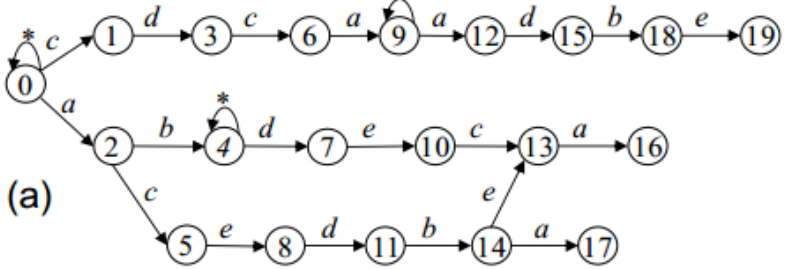
Orig. ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
New ID	0	10	4	13	1	11	12	14	17	2	18	15	5	3	8	16	6	7	9	19

	Orig. state ID	New state ID
overlap	0, 4, 9. 13	0, 1, 2, 3
a	2, 12, 16, 17	4, 5, 6, 7
b	14, 18	8, 9
c	1, 5, 6	10, 11, 12
d	3, 7, 11, 15	13, 14, 15, 16
e	8, 10, 19	17, 18, 19

Optimized NFA-based design

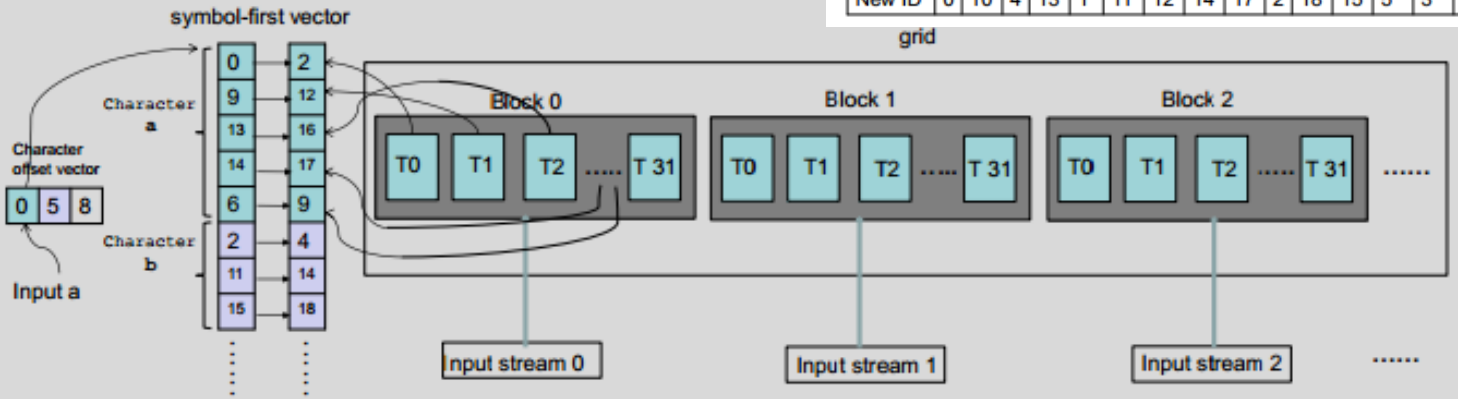


Optimized NFA-

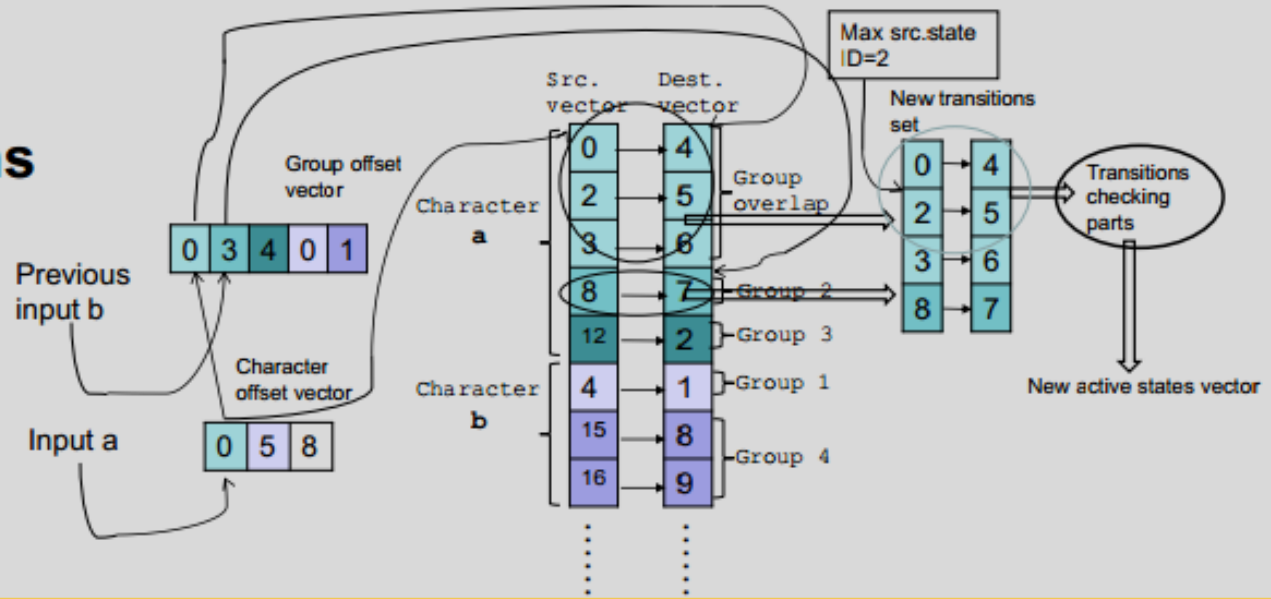


Orig. ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
New ID	0	10	4	13	1	11	12	14	17	2	18	15	5	3	8	16	6	7	9	19

iNFAnt implementation



Two optimizations applied to iNFAnt



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

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

Compressed DFA-based solution

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

```
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
```

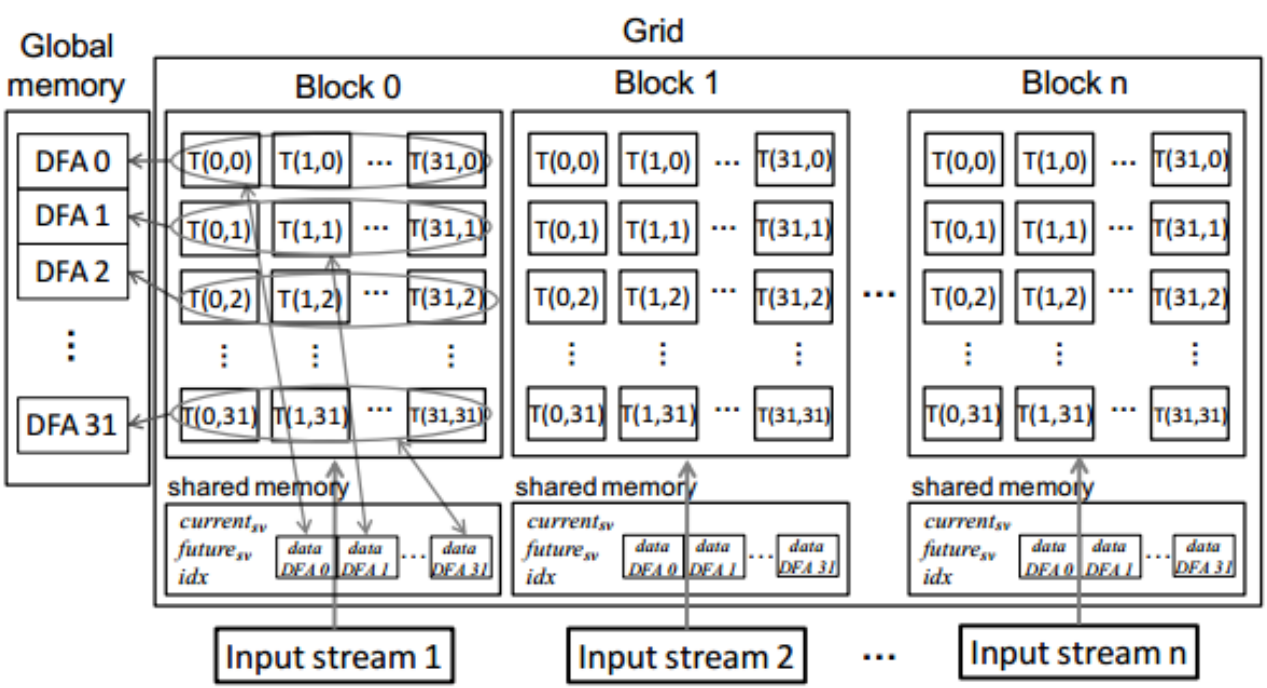
Compressed DFA-based solution

- 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

```
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
```



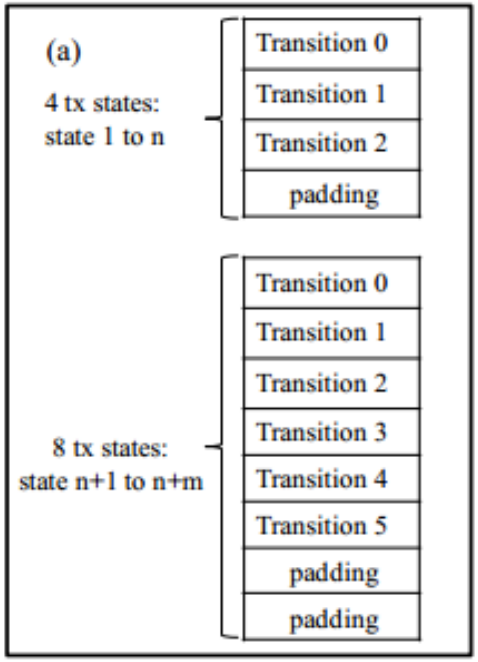
- After default transition compression, more than 90% of the states are left with 4-5 labeled transitions

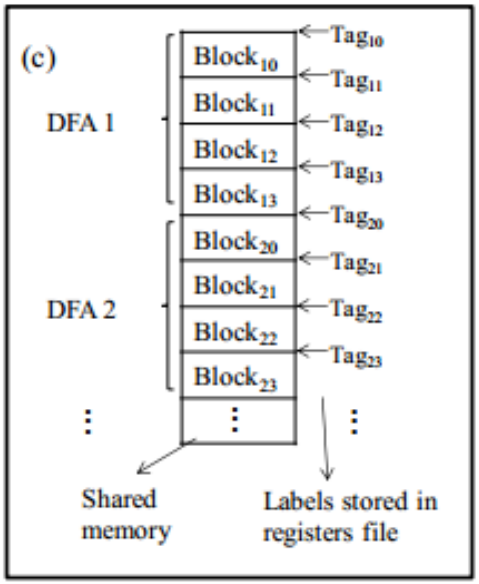
kernel compressed-DFA

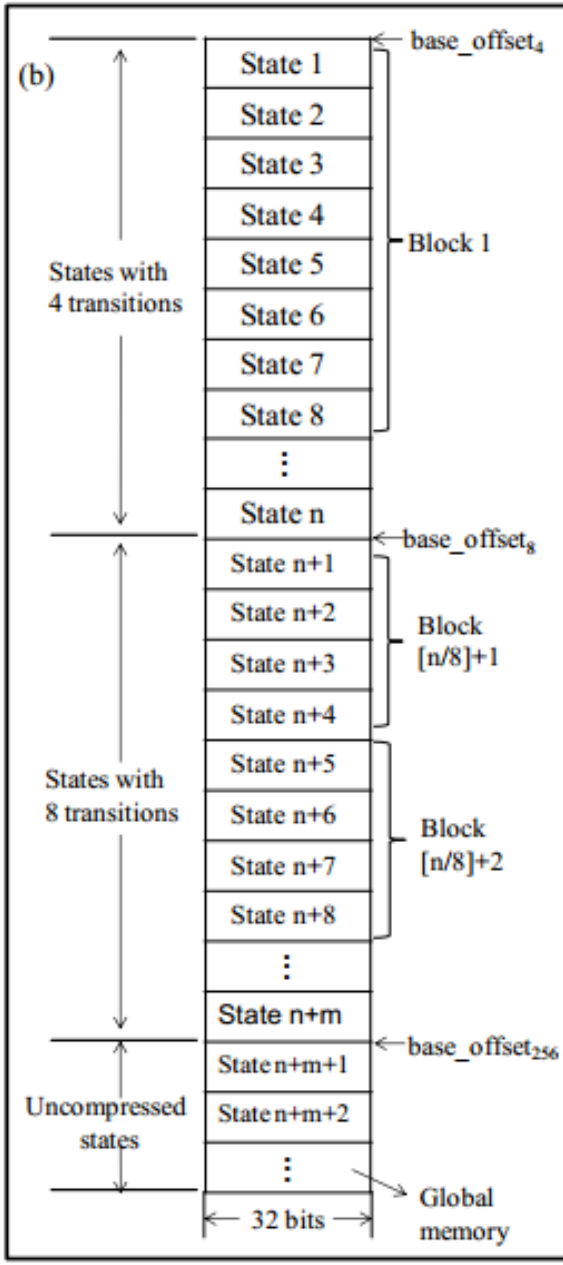
```
1:   current_sv[tid.y] ← initial_sv[tid.y]
2:   idx[tid.y] ← 0
3:   while (idx[tid.y]!=PACKET_SIZE) do
4:     future_sv[tid.y] ← INVALID
5:     c ← input[idx[tid.y]]
6:     tx_offset ← offset[current_sv[tid.y]]
7:     while current states has unprocessed transitions
8:       symbol ← labeled_tx[tid.y][tx_offset][it_offset+tid.x].char
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
```

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

Dataset	# regex	NFA			DFA							
		# states	# tx	Mem (MB)	# DFA	# states	Uncompressed-DFA		Compressed-DFA		Enhanced-DFA	
							# tx	Mem (MB)	# tx	Mem (MB)	# tx	Mem (MB)
Backdoor	226	4.3k	70.8k	0.54	13	960.1k	245.8M	942	20.27M	81	32M	126
Spyware	462	7.7k	66.8k	0.51	19	680.2k	174.1M	667	6.5M	27.5	21.6M	85
EM	1k	28.7k	51.9k	0.40	1	28.7k	7.35M	28.1	-	-	-	-
Range.5	1k	28.5k	91.8k	0.70	1	41.8k	10.7M	41	-	-	-	-
Range1	1k	29.6k	117.9k	0.90	1	54.4k	13.9M	53.3	-	-	-	-
Dotstar.05	1k	29.1k	116.8k	0.89	13	251k	64.26M	246	1.36M	6.2	2.35M	10
Dotstar.1	1k	29.2k	115.7k	0.88	21	603.8k	154.57M	592	3.37M	15.2	6.2M	26
Dotstar.2	1k	28.7k	114.6k	0.87	32	1.6M	418.1M	1,601	7.26M	33.9	15M	63.4

Performance evaluation

Dataset	CPU		GPU					CPU		GPU				
	NFA	DFA	NFA	O-NFA	U-DFA	C-DFA	E-DFA	NFA	DFA	NFA	O-NFA	U-DFA	C-DFA	E-DFA
	P_M=0.35							P_M=0.55						
Backdoor	0.4	7.9	40.8	37.6	171.4	13.8	42.6	0.5	7.7	39.9	37.1	169.1	13.8	42.6
Spyware	0.9	4.0	40.1	46.4	168.2	11.5	31.8	0.9	4.1	37.5	43.5	170.9	11.6	31.9
E-M	3.9	40.3	14.3	27.3	235.6	-	-	3.6	38.8	11.4	26.3	228.7	-	-
Range.5	4.9	40.6	13.6	26.7	227.1	-	-	4.1	39.0	12.5	25.7	225.5	-	-
Range1	4.9	41.1	18.6	25.7	211.8	-	-	4.1	39.4	15.4	24.7	219.9	-	-
Dotstar.05	1.4	7.9	20.0	25.8	190.6	13.7	37.1	1.0	7.7	17.1	24.5	183.4	13.6	37.1
Dotstar.1	0.9	5.2	18.7	26.0	158.1	8.8	30.2	0.7	5.8	19.8	21.4	156.8	8.7	30.6
Dotstar.2	0.6	3.2	18.6	24.0	-	6.2	26.3	0.6	3.4	21.7	22.9	-	6.1	27.9
	P_M=0.75							P_M=0.95						
Backdoor	0.4	7.4	37.2	31.9	166.8	13.7	40.3	0.2	7.4	35.8	30.4	149.1	13.6	39.2
Spyware	0.4	4.3	35.3	35.9	168.2	9.3	32.5	0.2	4.0	33.2	29.0	152.3	8.2	33.3
E-M	3.1	42.4	10.0	24.9	218.4	-	-	2.6	59.5	9.9	19.2	193.5	-	-
Range.5	3.3	42.3	16.9	24.2	208.3	-	-	2.5	57.2	14.0	18.1	186.6	-	-
Range1	3.3	42.0	13.3	23.3	209.1	-	-	2.5	53.6	10.2	17.2	186.6	-	-
Dotstar.05	0.5	7.6	13.2	17.6	150.8	13.4	38.8	0.2	5.0	13.1	15.7	153.8	12.5	40.4
Dotstar.1	0.6	5.2	16.4	20.8	156.1	8.1	29.1	0.1	3.3	13.1	14.4	137.0	7.5	31.4
Dotstar.2	0.8	3.0	14.6	19.4	-	6.0	26.3	0.02	1.9	12.8	13.3	-	5.8	28

Performance evaluation

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

Implementation	Optimal # flows per SM	Improvement over 1 flow per SM	
		Min	Max
U-DFA	5	1.72	2.75
C-DFA	5 (single-DFA) 2-3 (multi-DFAs)	1.16	2.82
E-DFA	5	2.49	3.33
iNFAnt	4	1.81	3.50
Opt-iNFAnt	4	2.55	3.65

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

Dataset	P_M=0.35		P_M=0.55		P_M=0.75		P_M=0.95	
	MR	PI	MR	PI	MR	PI	MR	PI
Backdoor	0.34	1.65	1.30	1.67	7.69	1.59	7.00	1.60
Spyware	0.59	1.52	3.47	1.53	8.06	1.41	17.91	1.36
Dotstar.05	2.77	1.66	8.35	1.54	35.14	1.09	37.44	0.98
Dotstar.1	2.81	1.37	7.49	1.40	15.40	1.25	36.05	1.16
Dotstar.2	6.07	1.16	4.44	1.15	10.16	1.10	39.57	0.93

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!