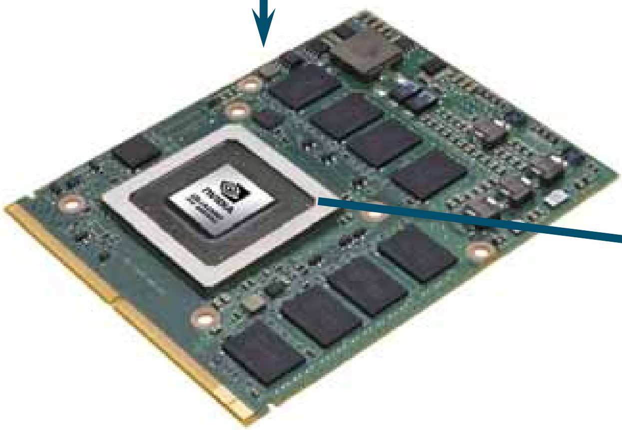
Electronic Design Automation (EDA) on GPUs

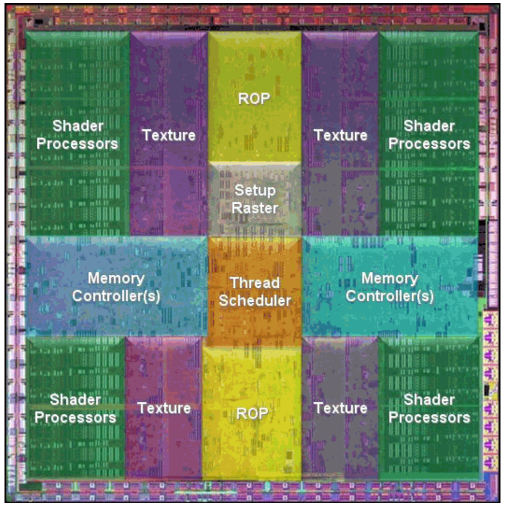
Yangdong Steve Deng
dengyd@tsinghua.edu.cn
Tsinghua University

Outline

- Introduction and motivation
- Sparse-matrix vector product (SMVP) on GPU
- GPU based logic simulation
- GPU based satisfiability solvers
- Conclusion and future work

Integrated Circuit – IC Chip

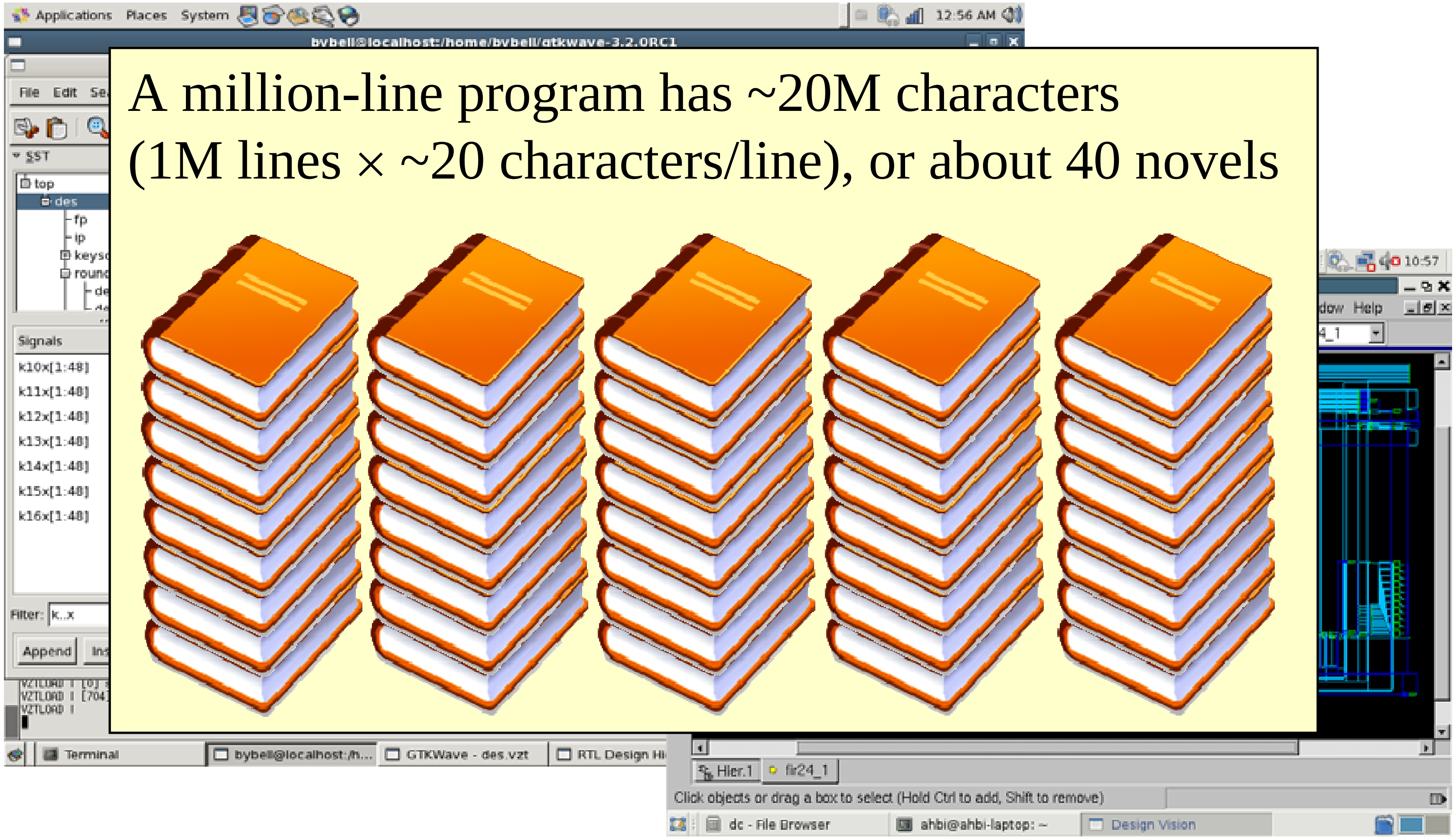




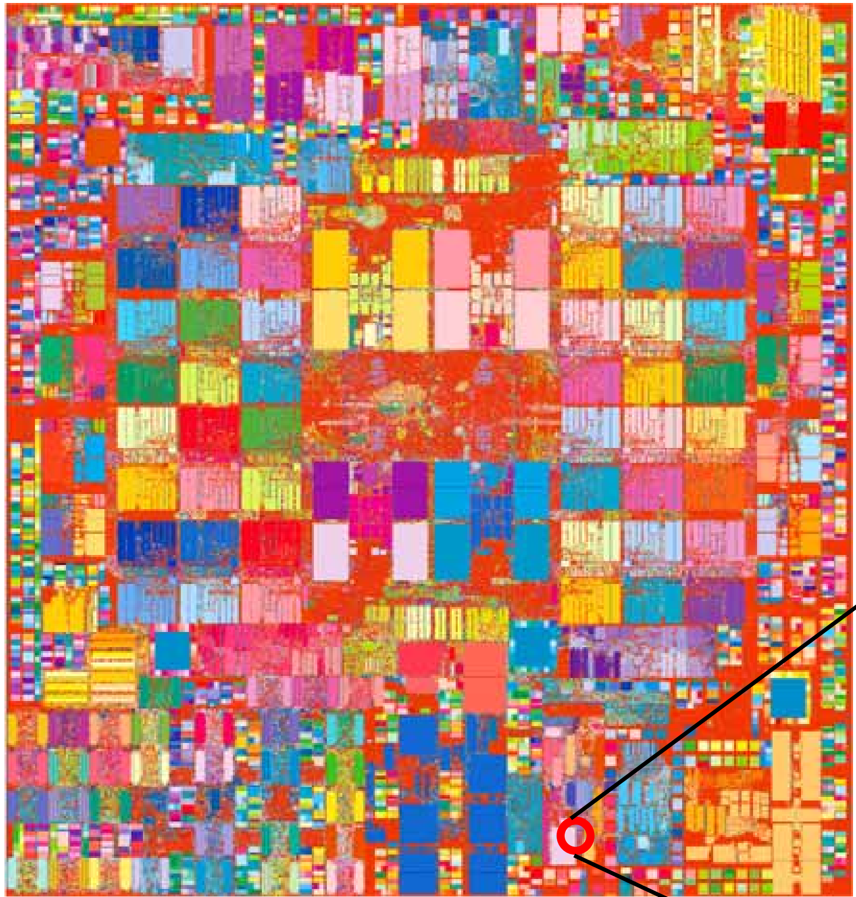
GPU Chip

IC Design Example – Logic Level

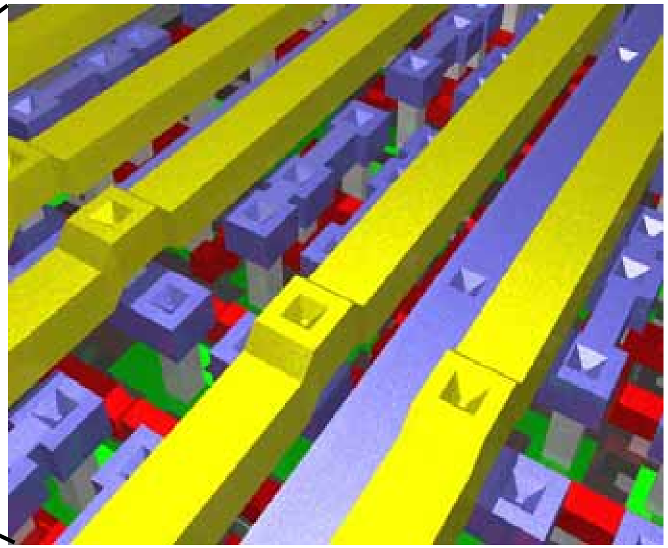
- 10M-gate design – 1M lines of code



IC Design Example - Layout

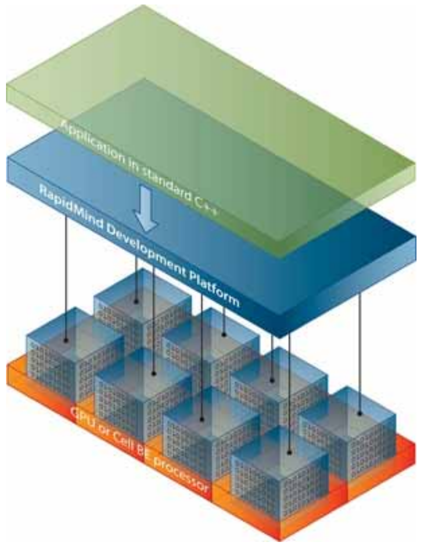


- Communication chip at 65nm
  - ~10M gates
  - ~1B timing paths
  - ~1T layout objects
  - 9 routing layers
  - 8 million nets
  - 1KM total wire length



Electronic Design Automation (EDA)

- Software to design complex IC chips
  - Overwhelming complexity
    - Escalating # of transistors, process variation, lithography printability, leakage, …
  - Skyrocketing tape-out cost
    - 45nm: $5~8M/mask set
  - Verification to ensure first-silicon success!
    - >70% design effort & >80% NRE cost
  - But single CPU performance saturates!
    - Parallelization is the only rescue!



Outline

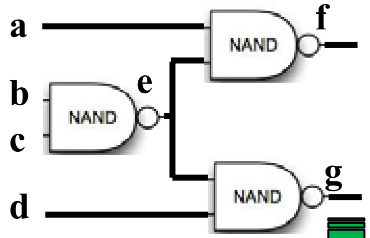
- Introduction and motivation
- Sparse-matrix vector product (SMVP) on GPU
- GPU based logic simulation
- GPU based satisfiability solvers
- Conclusion and future work

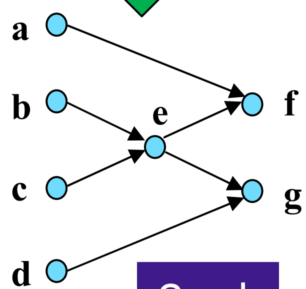
Irregularity of EDA Applications

- Regular data structures
  - e.g. linear array, dense matrix, …
- But EDA depends on irregular data structures
  - e.g. sparse matrix, graph, …
  - 12 out of 14 major EDA applications[1] involve sparse matrix and graph

$$\begin{bmatrix} 3 & 2 & 1 & 1 \\ 9 & 7 & 2 & 1 \\ 1 & 2 & 4 & 1 \\ 1 & 5 & 5 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 19 \\ 8 \\ 12 \end{bmatrix} \begin{matrix} \leftarrow \text{thread0} \\ \leftarrow \text{thread1} \\ \leftarrow \text{thread2} \\ \leftarrow \text{thread3} \end{matrix}$$

Matrix (Dense)





Graph

$$\begin{matrix} & a & b & c & d & e & f & g \\ a & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ b & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ d & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ f & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Matrix

$$rowptr = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 7 \quad 7 \quad 7]$$
$$col = [6 \quad 5 \quad 5 \quad 7 \quad 6 \quad 7]$$
$$elem = [1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1]$$

Sparse Matrix (CRS Format)

[1]Catanzaro, B., et al. *Parallelizing CAD: A Timely Research Agenda for EDA*. DAC08.

Sparse Matrix – Vector Product (SMVP)

- A major computing pattern for sparse matrix applications
  - Tend to be the performance bottleneck
  - e.g. >95% CPU time in our Conjugate Gradient solver
- Long considered as a tough case for parallel computing

$$p = Av = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 20 \\ 5 \end{bmatrix}$$

$$rowptr = \begin{bmatrix} 1 & 3 & 4 & 6 & 8 \end{bmatrix}$$

$$col = \begin{bmatrix} 1 & 3 & 2 & 3 & 4 & 1 & 4 \end{bmatrix}$$

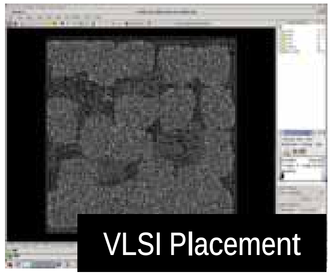
$$elm = \begin{bmatrix} 3 & 1 & 2 & 4 & 1 & 1 & 1 \end{bmatrix}$$

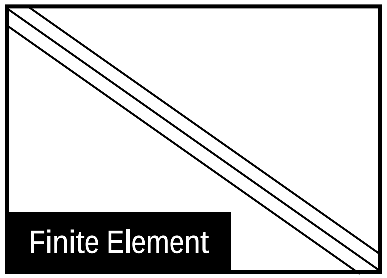
```
for(row = 1; row <= num_rows; row++){
    uint row_begin = rowptr[row];
    uint row_end = rowptr[row+1];
    float sum = 0.0;
    for(uint j= row_begin; j< row_end; ++j)
        sum += elem[j] * v[col[j]];
    p[row] = sum;
}
```

Prior Results of SMVP on GPU

- One thread for each row (#threads = #rows)
  - Experiment setup
    - CPU: 3.2GHz Core 2 Duo
    - GPU: GTX 8800
  - VLSI placement & finite element instances
    - Up to 1.2X speedup
    - But can even be slower on GPU
  - The Finite Element instances
    - 1.02 GFLOPS on GPU
    - 1.4X speedup against serial CPU version
- 32 treads for one row - Bell and Garland[2]
  - 10X speedup on certain problem instances
  - But only <5X on EDA problem instances

$$p = Av = \begin{bmatrix}3 & 0 & 0\\ 0 & 2 & 4\\ 1 & 0 & 0\end{bmatrix}\begin{bmatrix}1\\ 2\\ 3\end{bmatrix} = \begin{bmatrix}3\\ 16\\ 1\end{bmatrix}\begin{matrix}\leftarrow \text{thread0}\\ \leftarrow \text{thread1}\\ \leftarrow \text{thread2}\end{matrix}$$





[2]Bell, N. and Garland, M. *Efficient Sparse Matrix-Vector Multiplication on CUDA.* Supercomputing 2009.

Bottleneck

- Irregular memory access
  - >99% uncoalesced memory access
- Poor load-balance
  - Placement instances: #non-zeros in one row vary from 2 to 500

$$p = Av = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 20 \\ 5 \end{bmatrix}$$

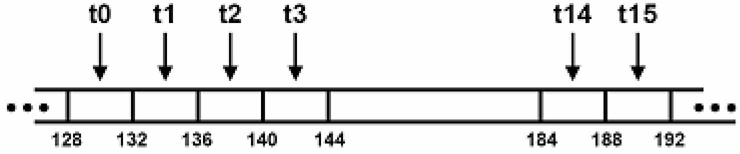
$$rowptr = \begin{bmatrix} 1 & 3 & 4 & 6 & 8 \end{bmatrix}$$

$$col = \begin{bmatrix} 1 & 3 & 2 & 3 & 4 & 1 & 4 \end{bmatrix}$$

$$elm = \begin{bmatrix} 3 & 1 & 2 & 4 & 1 & 1 & 1 \end{bmatrix}$$

```
uint row = blockIdx.x*blockDim.x + threadIdx.x;
if( row<num_rows ){
        uint row_begin = rowptr[row];
        uint row_end = rowptr[row+1];
        float sum = 0.0;
        for(uint j= row_begin; j< row_end; ++j)
            sum += elem[j] * v[col[j]];
        p[row] = sum;
}
```

Coalesced



Bad load balance

Generally non-coalesced

Reorganizing Data Flow in 2 Steps

- 1st step: element-wise product
  - middle[j] = elem[j] * v[col[j]] //Perfect load balance
  - Enhanced by an expansion procedure //Improved coalescing
- 2nd step: row-wise summation of element-wise products
  - sum = middle[row_begin] + … + middle[j] + … + middle[row_end]
  - Enhanced by a caching mechanism //Improved coalescing

```
if( row<num_rows ){
        uint row_begin = rowptr[row];
        uint row_end = rowptr[row+1];
        float sum = 0.0;
        for(uint j= row_begin; j< row_end; ++j)
            sum += elem[j] * v[col[j]];
        p[row] = sum;
}
```

$$p = Av = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 20 \\ 5 \end{bmatrix}$$

$$rowptr = [1 \quad 3 \quad 4 \quad 6 \quad 8]$$
$$col = [1 \quad 3 \quad 2 \quad 3 \quad 4 \quad 1 \quad 4]$$
$$elm = [3 \quad 1 \quad 2 \quad 4 \quad 1 \quad 1 \quad 1]$$

1. Element-wise product (the `*` in `elem[j] * v[col[j]]`)

2. Row-wise summation (the `+=` in `sum += …`)

SMVP Problem Instances

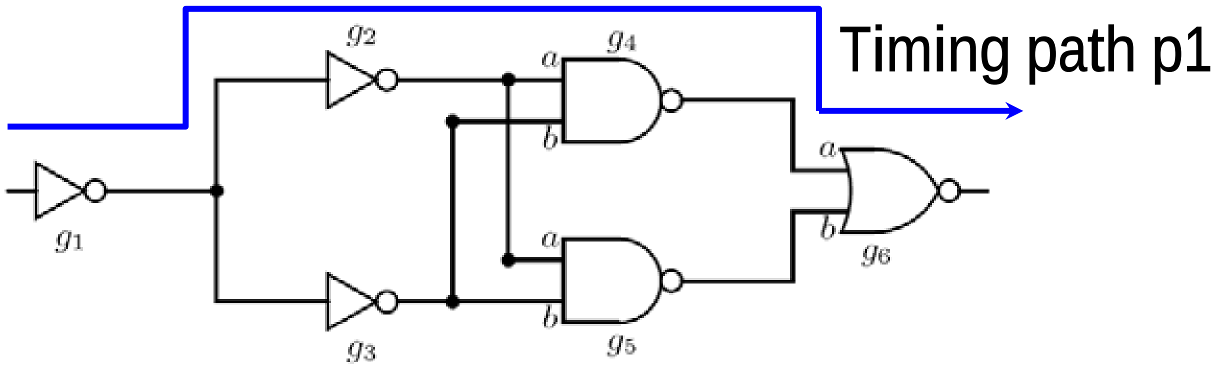
Problem Instance	# rows	# columns	# non-zeros	Avg. # non-zeros per row	Description
Lin	256000	256000	1766400	6.9	Large sparse Eigenvalue problem
t2em	921632	921632	4590832	5.0	Electromagnetic problems
ecology1	1000000	1000000	4996000	5.0	Circuit theory applied to animal/gene flow
cont11	1468599	1961394	5382999	3.7	Linear programming
sls	1748122	62729	6804304	3.9	Large least-squares problem
G3_circuit	1585478	1585478	7660826	4.8	AMD circuit simulation
thermal2	1228045	1228045	8580313	7.0	FEM, steady state thermal problem
kkt_power	2063494	2063494	12771361	6.2	Optimal power flow, nonlinear optimization
Freescale1	3428755	3428755	17052626	5.0	Freescale circuit simulation

SMVP Throughput on GTX280 GPU

Problem Instance	CPU (GFLOPS)	GPU (GFLOPS)	Speed-up
Lin	0.26	9.23	36.04
t2em	0.29	12.41	43.44
ecology1	0.24	9.03	37.43
cont11	0.31	10.66	33.84
sls	0.28	10.10	36.49
G3_circuit	0.21	8.86	41.45
thermal2	0.21	8.97	41.89
kkt_power	0.26	5.70	22.01
Freescale1	0.29	11.56	40.37

*Our CPU implementation is ~10% faster than Matlab

Application: Static Timing Analysis



$$\mathbf{A} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \begin{array}{c} \begin{array}{ccccccccc} g_1 & g_2 & g_3 & g_{4_a} & g_{4_b} & g_{5_a} & g_{5_b} & g_{6_a} & g_{6_b} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

$$\mathbf{d}_{\text{gate}} = \begin{bmatrix} d_1 & d_2 & d_3 & d_{4_a} & d_{4_b} & d_{5_a} & d_{5_b} & d_{6_a} & d_{6_b} \end{bmatrix}^\top \qquad \mathbf{d}_{\text{path}} = \mathbf{A}\mathbf{d}_{\text{gate}}$$

Adapted from Ramalingam, A. et. al. *An Accurate Sparse Matrix Based Framework for Statistical Static Timing Analysis*. ICCAD. 2006.

Static Timing Analysis Results on GTX280

Instance	CPU (#paths per second)	GPU (#paths per second)	Speed-up
b18_50K	1.95E+06	1.02E+08	52.33
b18_100K	1.92E+06	9.57E+07	49.82
b19_50K	2.46E+06	1.10E+08	44.64
b19_100K	2.42E+06	1.14E+08	47.00

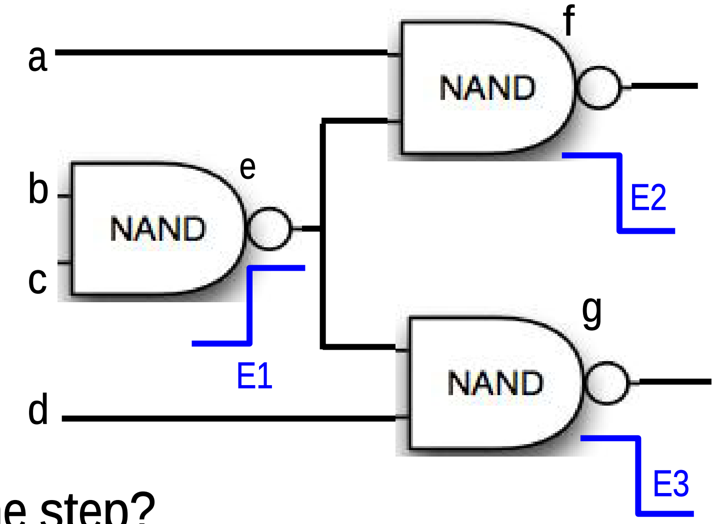
1M-gate ASIC: 60 min@CPU vs. 1 min@GPU

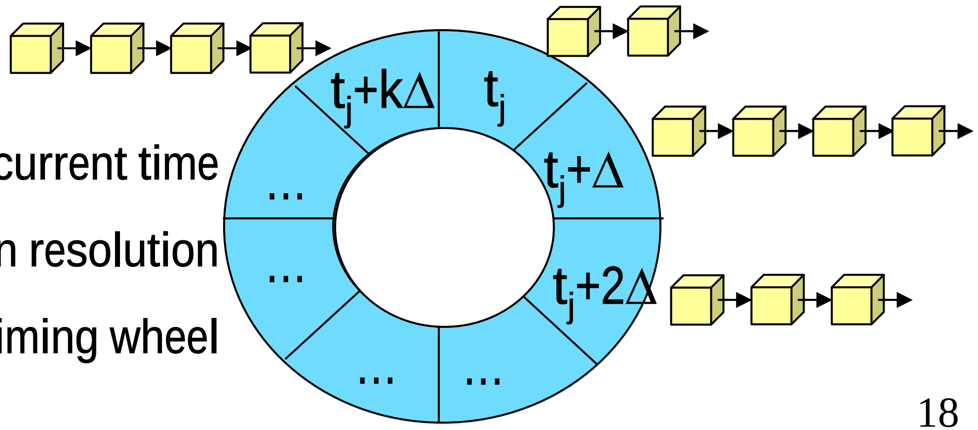
Outline

- Introduction and motivation
- Sparse-matrix vector product (SMVP) on GPU
- GPU based logic simulation
- GPU based satisfiability solvers
- Conclusion and future work

Logic Simulation

- Essential EDA tool for verification
- Event Driven logic simulation
  - Store events in a queue
  - Ordered by a global clock
  - Pick up the top event to simulate
- How to get parallel?
  - Executing operations at the same time step?
  - Very limited parallelism!





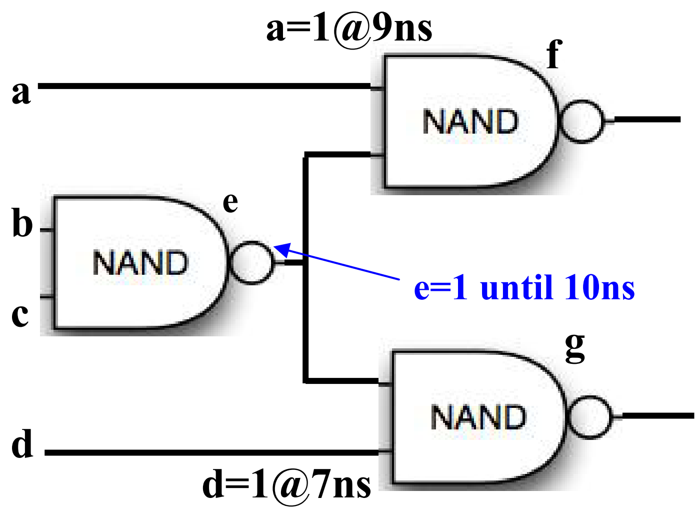
t_j: current time

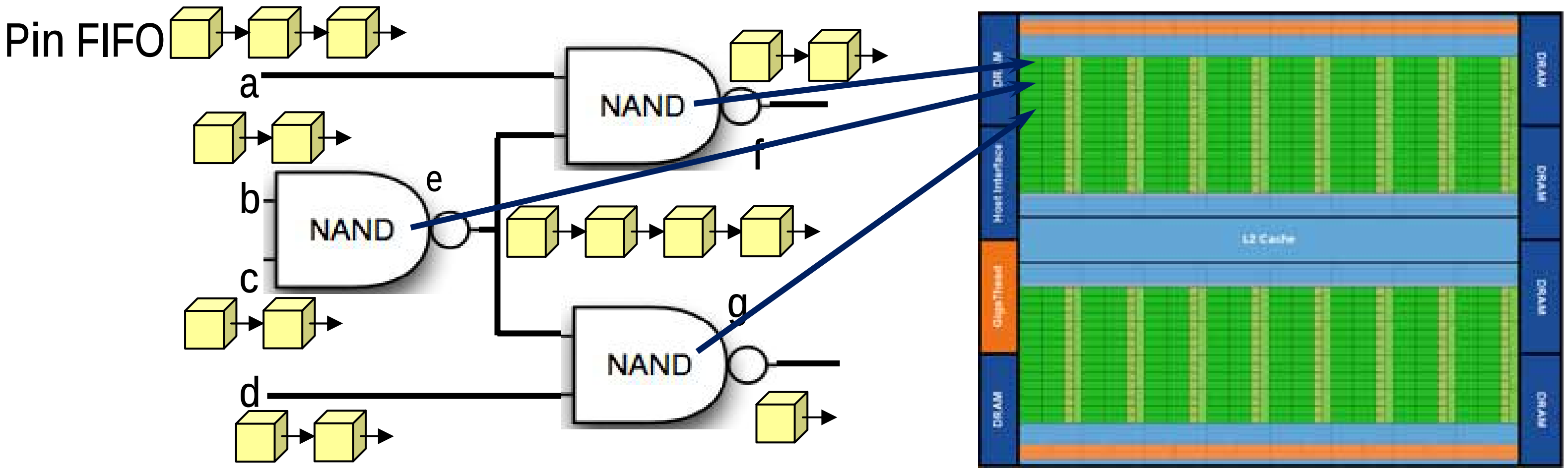
Δ: minimum simulation resolution

L+1: number of entries in the timing wheel

Chandy-Misra-Bryant (CMB) on GPUs

- Distributed-time logic simulation
  - One thread for one gate
  - Every pin maintains a local FIFO for events
    - FIFO sizes vary dramatically
  - A dynamic GPU memory manager
    - Garbage collection and recycle
  - Deadlock prevention





Results - Deterministic Test Patterns

- 30X speed-up on average (100X for random patterns)
  - 1 month on CPU vs. 1 day on GPU

Design	#gates	Simulated cycles	CPU Simulation time (s)	GPU Simulation time (s)	Speed-up
AES	13,118	42,935,000	109.90	4.45	24.70
DES3	53,131	30,730,000	183.11	4.50	40.66
SHA1	5,616	2,275,000	56.66	0.41	138.20
JPEG	117,701	26,132,000	136.33	43.09	3.16
NOC	64,095	1,000,000	5389.42	347.95	15.49
M1	14,850	99,998,019	118.48	22.43	5.28

*Our CPU implementation is ~40% faster than Synopsys VCS

Outline

- Introduction and motivation
- Sparse-matrix vector product (SMVP) on GPU
- GPU based logic simulation
- GPU based satisfiability solvers
- Conclusion and future work

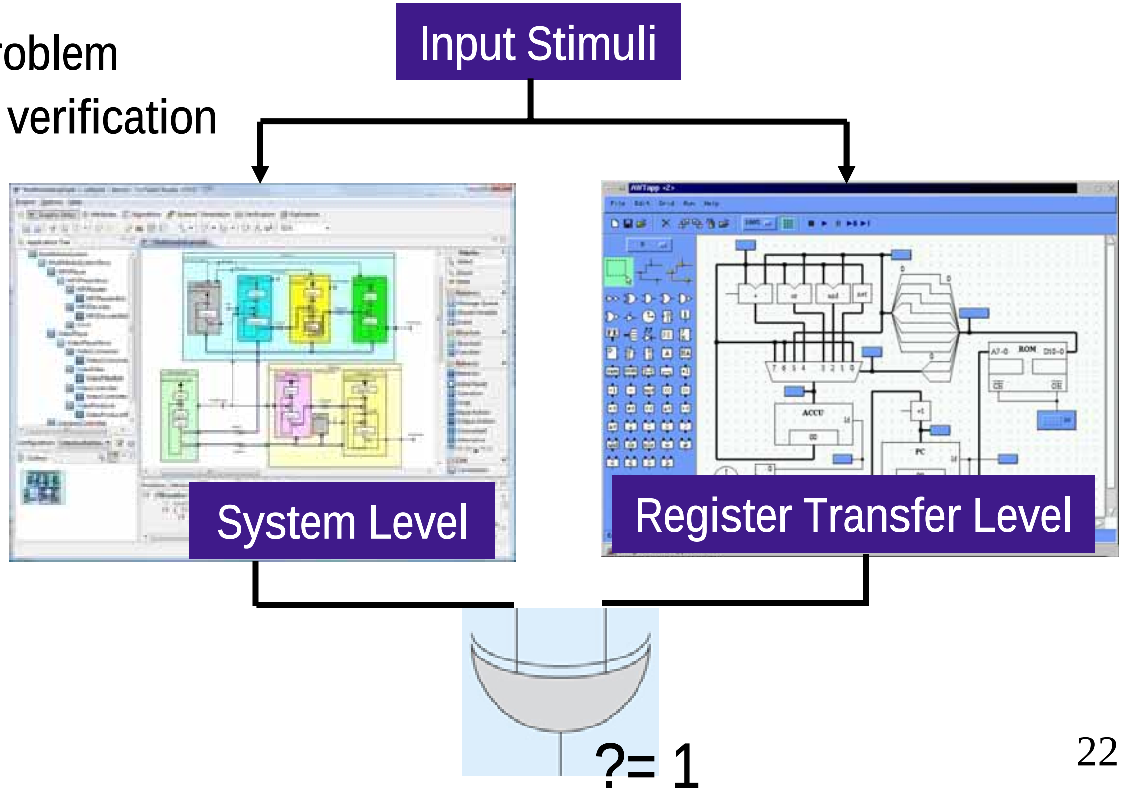
Satisfiability (SAT) Problem

- Given a Boolean propositional formula, determine whether there exists a variable assignment that makes the formula evaluate to true.

$$(x1+x2+\neg x3) \wedge (\neg x1+x3+x4) \wedge (\neg x2+x3+\neg x4)$$

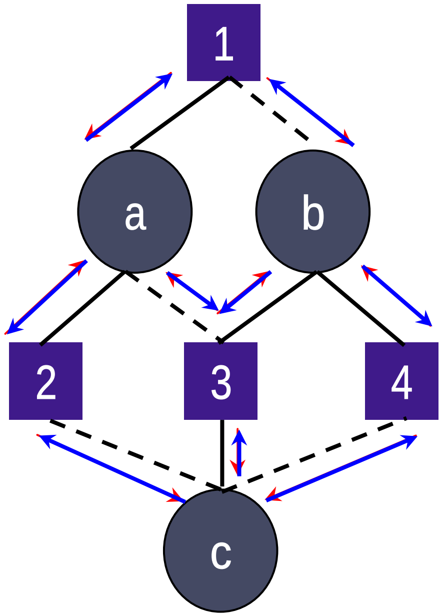
Clause *Literal* *Variables: x1, x2, x3, and x4*

- 1st NP-Complete problem
- Core engine formal verification



Random Algorithm - Survey Propagation

- Iterative updating through message passing
  - Originated from statistical physics
  - Good at "hard" problem instances



$$\Pi^u_{j\to a}(t) = \left[1 - \prod_{b\in C^u_a(j)} (1-\eta^t_{b\to j})\right] \prod_{b\in C^s_a(j)} (1-\eta^t_{b\to j})$$

$$\Pi^s_{j\to a}(t) = \left[1 - \prod_{b\in C^s_a(j)} (1-\eta^t_{b\to j})\right] \prod_{b\in C^u_a(j)} (1-\eta^t_{b\to j})$$

$$\Pi^0_{j\to a}(t) = \prod_{b\in C(j)\setminus a} (1-\eta^t_{b\to j})$$

Message from variable to clause

$$\eta^{t+1}_{a\to j} = \prod_{j\in V(a)\setminus i} \left[\frac{\Pi^u_{j\to a}(t)}{\Pi^u_{j\to a}(t) + \Pi^s_{j\to a}(t) + \Pi^0_{j\to a}(t)}\right]$$

Message from clause to variable

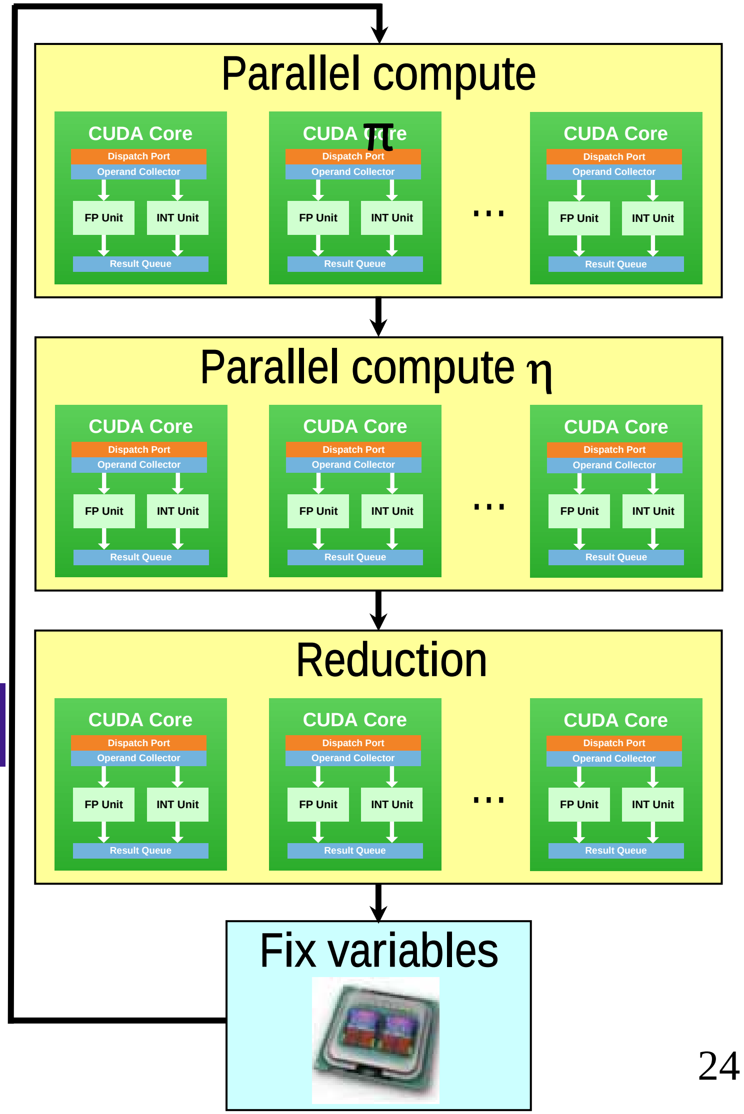
$$\underbrace{(x1+x2+\neg x3)}_{a}\wedge\underbrace{(\neg x1+x3+x4)}_{b}\wedge\underbrace{(\neg x2+x3+\neg x4)}_{c}$$

Survey Propagation on GPUs

- Parallel message passing

#variables	CPU(s)	GPU(s)	Speed-up
50,000	45.37	2.32	19.6
100,000	116.34	5.55	21.0
300,000	404.69	19.57	20.7
500,000	710.76	43.31	20.7
900,000	1415.29	66.97	21.1

SAT 2009 competition random benchmark



Outline

- Introduction and motivation
- Sparse-matrix vector product (SMVP) on GPU
- GPU based logic simulation
- GPU based satisfiability solvers
- Conclusion and future work

Conclusion and Future Work

- A systematic effort to accelerate EDA computing on GPUs
  - Sparse matrix vector product (20-50X speedup)
    - Static timing analysis (50X)
  - Logic simulation (30-130X)
  - Survey propagation based SAT solver (20X)
- Future work
  - Circuit simulation (SPICE)
    - Matrix assembling + direct method
  - A heterogeneous SAT solver
    - CPU: DPLL solver
    - GPU: Local search solver
    - Exchange solution information