Accelerating Statistical Static Timing Analysis Using Graphics Processing Units

Kanupriya Gulati & Sunil P. Khatri
Department of ECE
Texas A&M University, College Station

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

Introduction

- Static timing analysis (STA) is heavily used in VLSI design to estimate circuit delay
- Impact of process variations on circuit delay is increasing
- Therefore, ***statistical STA (SSTA)*** was proposed
  - It includes the effect of variations while estimating circuit delay
- ***Monte Carlo (MC) based SSTA*** accounts for variations by
  - Generating *N* delay samples for each gate (random variable)
  - Executing STA for each sample
  - Aggregating results to generate full circuit delay under variations
- MC based SSTA has several advantages (over Block based SSTA)
  - High accuracy, simplicity and compatibility to fabrication line data
- Main disadvantage is ***extremely high runtime cost***

Introduction

- We accelerate MC based SSTA using graphics processing units (GPUs)
- A GPU is essentially a commodity stream processor
  - Highly parallel
  - Very fast
  - SIMD (Single-Instruction, Multiple Data) operating paradigm
- GPUs, owing to their **massively parallel architecture**, have been used to accelerate several **scientific computations**
  - Image/stream processing
  - Data compression
  - Numerical algorithms
    - LU decomposition, FFT etc

Introduction

- We implemented our approach on the
  - NVIDIA ***GeForce 8800 GTX GPU***
- By careful engineering, we maximally harness the GPU's
  - Raw computational power and
  - Huge memory bandwidth
- Used Compute Unified Device Architecture (CUDA) framework
  - Open source GPU programming and interfacing tool
- When using a *single* 8800 GTX GPU card
  - ***~260X speedup*** in MC based SSTA is obtained
  - Accounts for CPU processing and data transfer times as well
- The SSTA runtimes are projected for a Quad GPU system
  - NVIDIA SLI technology allows 4 GPU devices on the same board: ***~788X speedup*** is possible

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

GeForce 8800 GTX Technical Specs.

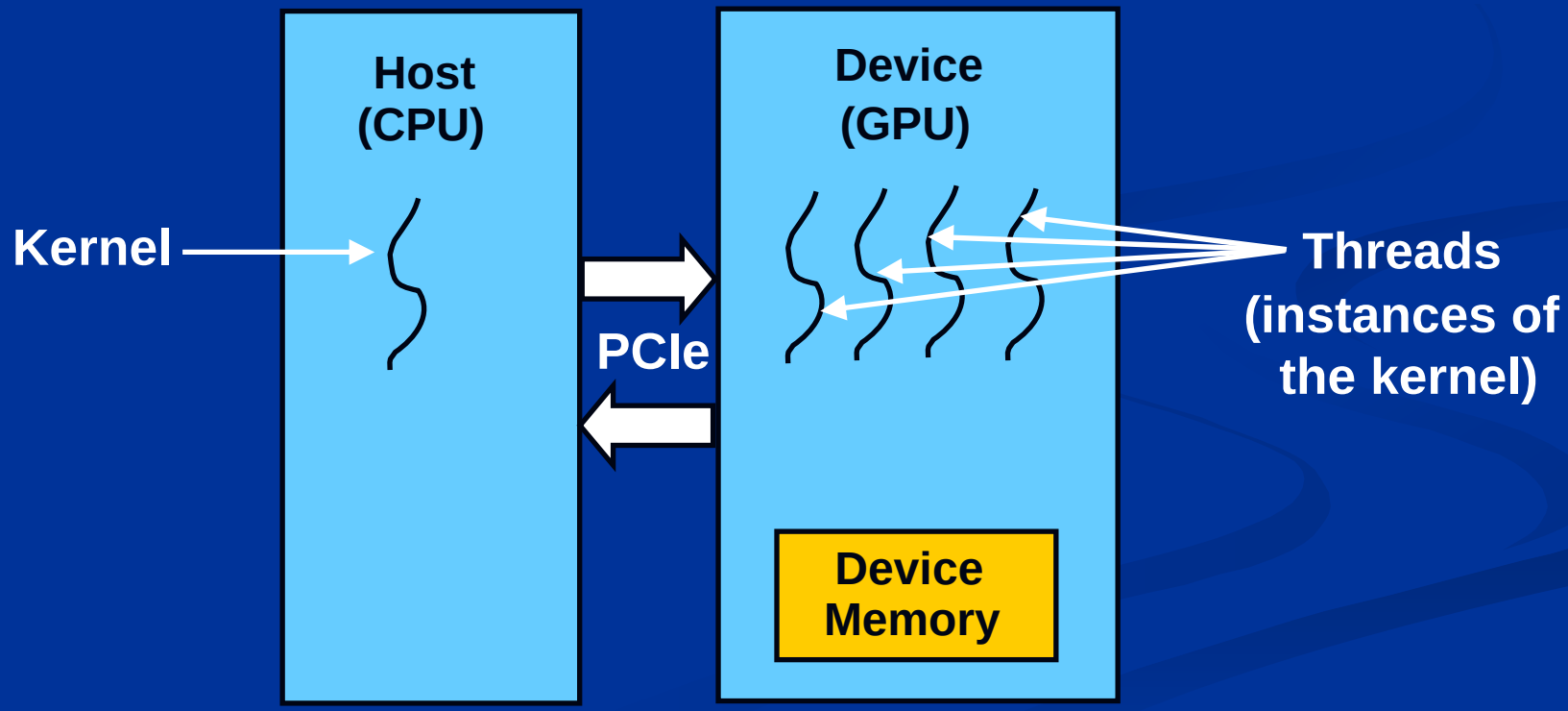
- 367 GFLOPS peak performance
  - 25-50 times of current high-end microprocessors
- 265 GFLOPS sustained for appropriate applications
- Massively parallel, 128 cores
  - Partitioned into 16 Multiprocessors
- Massively threaded, sustains 1000s of threads per application
- 768 MB device memory
- 1.4 GHz clock frequency
  - CPU at 3.6 GHz
- 86.4 GB/sec memory bandwidth
  - CPU at 8 GB/sec front side bus
- Multi-GPU servers available
  - SLI Quad high-end NVIDIA GPUs on a single motherboard
  - Also 8-GPU servers announced recently

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

CUDA Programming Model

- The GPU is viewed as a compute **device** that:
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**

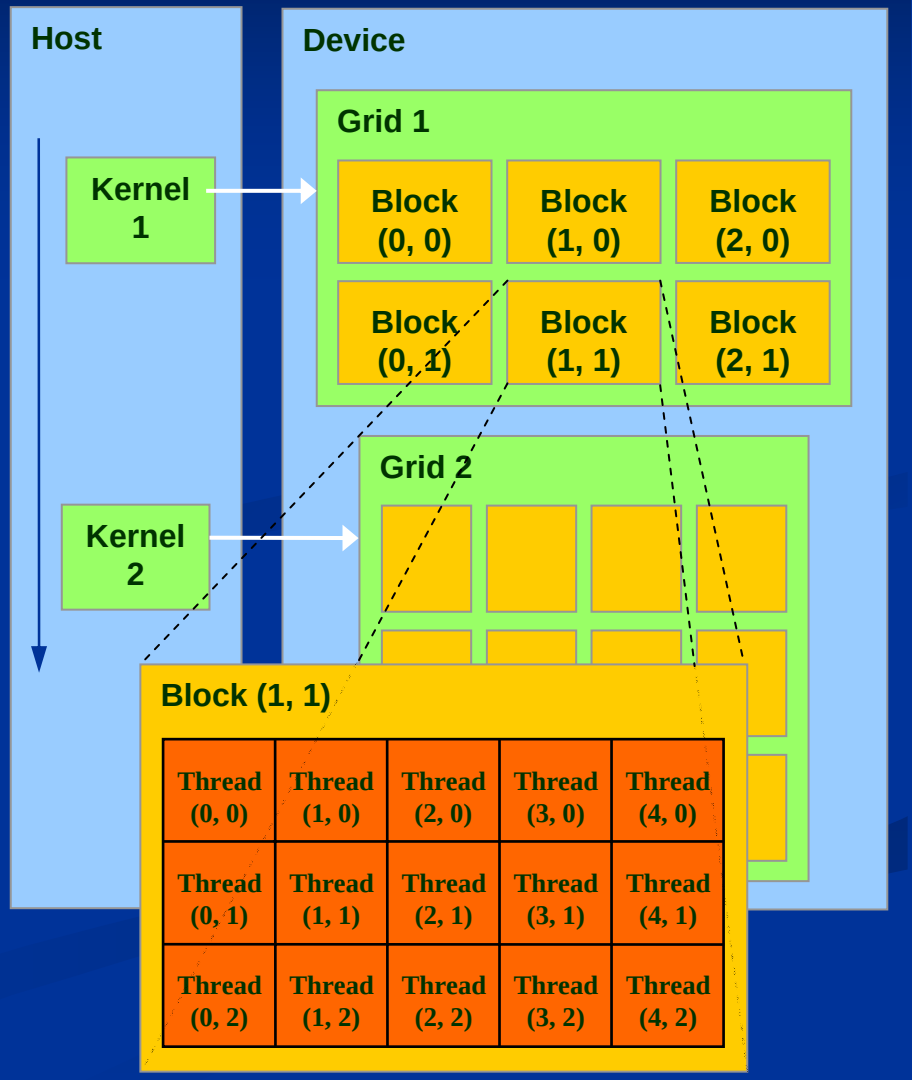


CUDA Programming Model

- Data-parallel portions of an application are executed on the device in parallel on many threads
  - ***Kernel*** : code routine executed on GPU
  - ***Thread*** : instance of a kernel
- Differences between GPU and CPU threads
  - GPU threads are lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads to achieve full parallelism
    - Allows memory access latencies to be hidden
    - Multi-core CPUs require fewer threads, but the available parallelism is lower

Thread Batching: Grids and Blocks

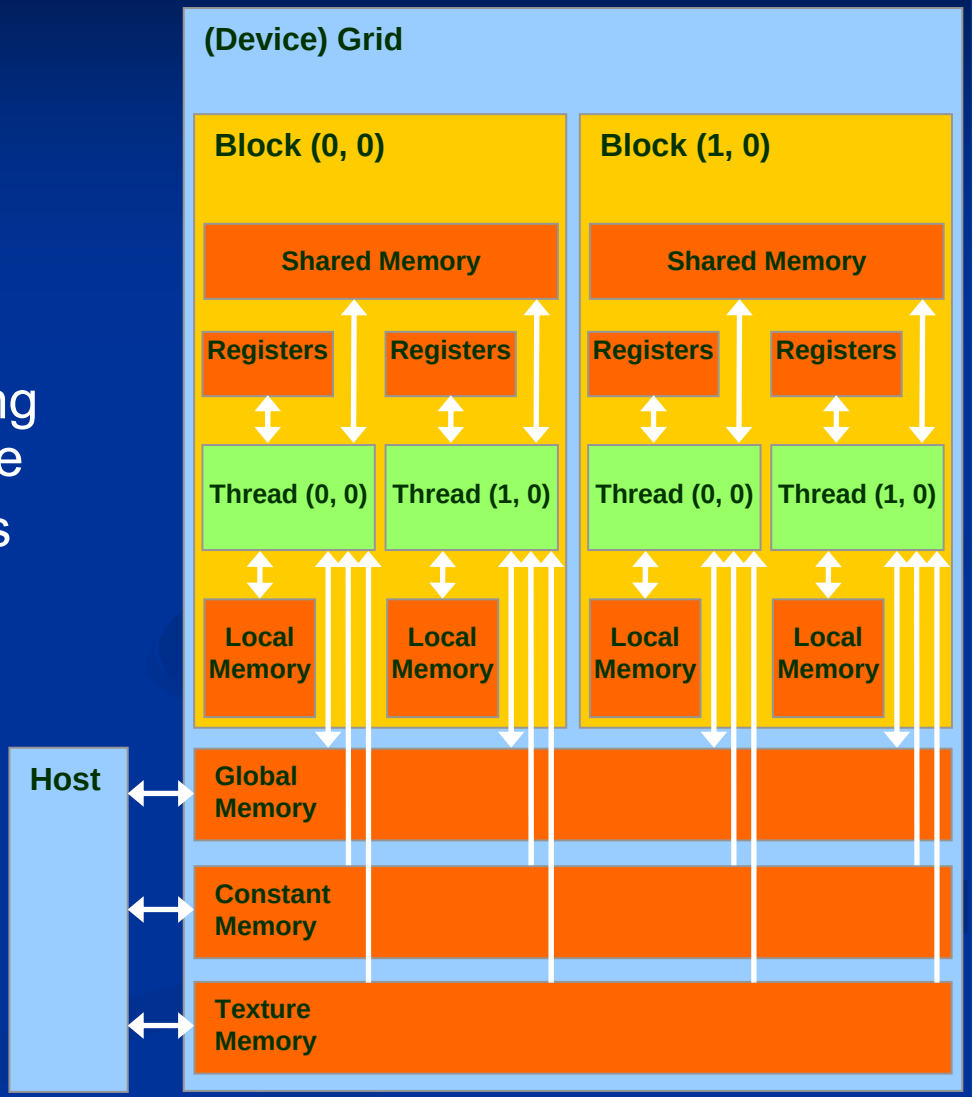
- A kernel is executed as a **grid of thread blocks (aka blocks)**
  - All threads within a block share a portion of data memory
  - Threads/blocks have 1D/2D/3D IDs
- A **thread block** is a batch of threads that can **cooperate** with each other by:
  - Synchronizing their execution
    - For hazard-free common memory accesses
  - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot cooperate



Source : "NVIDIA CUDA Programming Guide" version 1.1

Device Memory Space Overview

- Each thread has:
  - R/W per-thread **registers** (max. 8192 registers/MP)
  - R/W per-thread **local memory**
  - R/W per-block **shared memory**
  - R/W per-grid **global memory**
    - Main means of communicating data between host and device
    - Contents visible to all threads
    - Coalescing recommended
  - Read only per-grid **constant memory**
    - Cached, visible to all threads
  - Read only per-grid **texture memory**
    - Cached, visible to all threads
- The host can R/W **global**, **constant** and **texture** memories



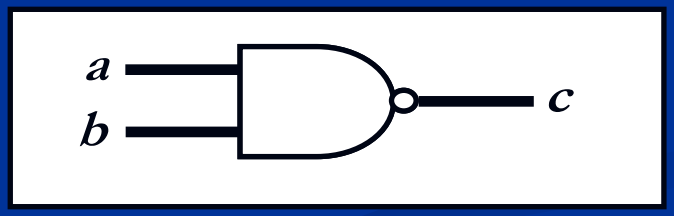
Source : "NVIDIA CUDA Programming Guide" version 1.1

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

Approach - STA

- **STA at a gate**
  - Over all inputs compute the ***MAX*** of
    - ***SUM*** of
      - **Input arrival time for input *i* and**
      - **Pin-to-output (P-to-O) rising (or falling) delay from pin *i* to output**



  - For example, let
    - AT_i^{fall} denote the arrival time of a *falling signal* at *node i*
    - AT_i^{rise} denote the arrival time of a *rising signal* at *node i*

$$AT_c^{rise} = MAX\,[(AT_a^{fall} + MAX\,(D_{11\rightarrow 00}, D_{11\rightarrow 01})),\ (AT_b^{fall} + MAX\,(D_{11\rightarrow 00}, D_{11\rightarrow 10}))]$$

$MAX\,(D_{11\rightarrow 00}, D_{11\rightarrow 01})$ denotes the P-to-O rising delay from *a* to *c*

$MAX\,(D_{11\rightarrow 00}, D_{11\rightarrow 10})$ denotes the P-to-O rising delay from *b* to *c*

Approach - STA

- In order to implement STA at a gate, on the GPU
  - The P-to-O rising (or falling) delay from every input to output is stored in a lookup table (LUT)
    - ***LUT stored in texture memory*** of GPU. Key advantages:
      - Cached memory: LUT data easily fits into available cache
      - No memory coalescing requirements
      - Efficient built-in texture fetching routines available in CUDA
      - Non-zero time taken to load texture memory, but cost amortized
  - For an *n*-input gate, do the following
    - ***Fetch n pin-to-output*** rising (or falling) delays from texture memory
      - Using the gate type offset, pin number and falling/rising delay information
    - ***n SUM computations*** (of the pin-to-output delay and input arrival time)
    - ***n-1 MAX computations*** (since CUDA only supports 2 operand MAX operations)

Approach - STA

```
typedef struct __align__(8){
int offset; //Gate type's offset
float a, b, c, d; // Arrival times for the inputs
} threadData;
```

Algorithm 1 Pseudocode of the kernel for rising output STA for inverting gate

$static\_timing\_kernel(threadData * MEM, float * DEL)\{$
$t_x = my\_thread\_id$
$threadData\ Data = MEM[t_x]$
$p2pdelay\_a = tex1D(LUT, MEM[t_x].offset + 2 \times 0);$
$p2pdelay\_b = tex1D(LUT, MEM[t_x].offset + 2 \times 1);$
$p2pdelay\_c = tex1D(LUT, MEM[t_x].offset + 2 \times 2);$
$p2pdelay\_d = tex1D(LUT, MEM[t_x].offset + 2 \times 3);$
$LAT = fmaxf(MEM[t_x].a + p2pdelay\_a, MEM[t_x].b + p2pdelay\_b);$
$LAT = fmaxf(LAT, MEM[t_x].c + p2pdelay\_c);$
$DEL[t_x] = fmaxf(LAT, MEM[t_x].d + p2pdelay\_d);$
$\}$

Approach - SSTA

- **SSTA at a gate**
  - Need (μ , σ) for the *n* Gaussian distributions of the pin-to-output rising and falling delay values for *n* inputs
  - ***Store (μ , σ) for every input*** in the LUT
    - As opposed to storing the nominal delay
  - ***Mersenne Twister (MT)*** pseudo random number generator is used. It has the following advantages
    - Long period
    - Efficient use of memory
    - Good distribution properties
    - Easily parallelizable (in a SIMD paradigm)
  - The uniformly distributed random number sequences are then transformed into the normal distribution N(0,1)
    - Using the ***Box-Muller transformations (BM)***
  - Both algorithms, MT and BM are implemented as separate kernels

Approach - SSTA

Algorithm 2 Pseudocode of the kernel for rising output SSTA for inverting gate

$statistical\_static\_timing\_kernel(threadData * MEM, float * DEL)\{$
$t_x = my\_thread\_id$
$threadData\ Data = MEM[t_x]$
$p2pdelay\_a^{\mu} = tex1D(LUT^{\mu}, MEM[t_x].offset + 2 \times 0);$
$p2pdelay\_a^{\sigma} = tex1D(LUT^{\sigma}, MEM[t_x].offset + 2 \times 0);$
$p2pdelay\_b^{\mu} = tex1D(LUT^{\mu}, MEM[t_x].offset + 2 \times 1);$
$p2pdelay\_b^{\sigma} = tex1D(LUT^{\sigma}, MEM[t_x].offset + 2 \times 1);$
$p2pdelay\_c^{\mu} = tex1D(LUT^{\mu}, MEM[t_x].offset + 2 \times 2);$
$p2pdelay\_c^{\sigma} = tex1D(LUT^{\sigma}, MEM[t_x].offset + 2 \times 2);$
$p2pdelay\_d^{\mu} = tex1D(LUT^{\mu}, MEM[t_x].offset + 2 \times 3);$
$p2pdelay\_d^{\sigma} = tex1D(LUT^{\sigma}, MEM[t_x].offset + 2 \times 3);$
$p2p\_a = p2pdelay\_a^{\mu} + k_a \times p2pdelay\_a^{\sigma}$; // k_a, k_b, k_c, k_d
$p2p\_b = p2pdelay\_b^{\mu} + k_b \times p2pdelay\_b^{\sigma}$; // are obtained by Mersenne
$p2p\_c = p2pdelay\_c^{\mu} + k_c \times p2pdelay\_c^{\sigma}$; // Twister followed by
$p2p\_d = p2pdelay\_d^{\mu} + k_d \times p2pdelay\_d^{\sigma}$; // Box-Muller transformations.
$LAT = fmaxf(MEM[t_x].a + p2p\_a, MEM[t_x].b + p2p\_b);$
$LAT = fmaxf(LAT, MEM[t_x].c + p2p\_c);$
$DEL[t_x] = fmaxf(LAT, MEM[t_x].d + p2p\_d);$
$\}$

Approach - SSTA

- For a circuit, SSTA is performed topologically from inputs to outputs
  - Delays of gates at logic depth *i* are computed, and stored in global memory
  - Gates at logic higher depths *may* use this data as their input arrival times
- GPU performance is maximized by ensuring that:
  - **Data dependency** between threads issued in parallel is avoided
  - Threads issued in parallel execute same instruction, but on different data
    - Conforms to the **SIMD architecture** of GPUs
  - **SIMD-based implementation for MT pseudorandom number generator** is used
  - Specific to G80 architecture
    - **Texture Memory** is used for storing LUT for μ and σ values
    - Global memory writes for level *i* gates (and reads for level *i+1* gates) are performed in a **coalesced fashion**
- Can be easily extended for Statistical Timing Analysis with spatial correlations
  - Existing approaches to implement principal component analysis (PCA) in a SIMD fashion

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

Experiments

- MC based SSTA on 8800 GTX runtimes compared to a CPU based implementation
  - 30 large IWLS and ITC benchmarks.
- Monte Carlo analysis performed by using 64 K samples for all 30 circuits.
- CPU runtimes are computed
  - On 3.6 GHz, 3GB RAM Intel processor running Linux.
  - Using *getrusage* (system + user) time
- GPU (wall clock) time computed using CUDA on GeForce 8800 GTX

Experiments

- ***GPU time includes data transfer time*** GPU ↔ CPU
  - CPU → GPU :
    - arrival time at each primary input
    - μ and σ for all pin-to-output delays of all gates
  - GPU → CPU:
    - 64K delay values at each primary output
- GPU times also ***include the time spent in the MT and BM kernels, and loading texture memory***
- Computation results have been verified for correctness
- For the SLI Quad system, the runtimes are obtained by scaling the processing times only
  - Transfer times are included as well (not scaled)

Results

Circuit	Runtime (s)			Speedup	
	GPU	SLI Quad	CPU	GPU	SLI Quad
s9234_1	1.949	0.672	499.981	256.570	744.307
s35932	11.318	4.341	2731.638	241.349	629.197
s38584	11.544	4.163	2889.924	250.335	694.158
s13207	3.512	1.461	802.963	228.663	549.517
:	:	:	:	:	:
b22_1	12.519	3.665	3466.783	276.913	945.897
b21	10.311	2.956	2879.765	279.298	974.323
b15_1	6.952	2.121	1891.884	272.116	89.174
Avg. (30 Ckts.)				**258.994**	**788.014**

- Using a single GPU, the speedup over CPU is ~260X
- Projecting to SLI Quad shows speedup of ~788X
  - Recently an 8-GPU NVIDIA Tesla server has been announced
- Block based SSTA can achieve ~500X [Le et al DAC 04] speedup over MC based SSTA
  - However, they report a 2% error compared to MC based SSTA

Outline

- Introduction
- Technical Specifications of the GPU
- CUDA Programming Model
- Approach
- Experimental Setup and Results
- Conclusions

Conclusions

- STA is used in VLSI design flow to estimate circuit delay
- Process variations are growing larger and less systematic
- MC based SSTA accounts for variations, and has several advantages like high accuracy and compatibility to data obtained from the fab line
- Main disadvantage is extremely high runtime cost
- We accelerate MC based SSTA using graphics processing units (GPUs)
- By careful engineering, we maximally harness the GPU's
  - ***Raw computational power and***
  - ***Huge memory bandwidths***
- When using a Single 8800 GTX GPU
  - ***~260X speedup*** in MC based SSTA is obtained
- The SSTA runtimes are projected on a Quad GPU system
  - ***~785X speedup*** is possible

Thank You!