Monte Carlo Estimation of pi using GPU

Shriya Nair

The University of Texas at Austin

December 9, 2016

Overview . . .

- Problem Statement
- Overview of GPU
- GPU Implementation and Optimizations
- Overview of Features
- Previous Implementations
- Results
  - GPU vs CPU
  - Thrust Library vs My implementation
- github Structure and How Tos
- Build System
- Scripts
- What would I have done differently?
- Conclusion

Problem Statement

The problem statement is to answer the following questions:

- How fast is GPU Implementation compared to Serial implementation?
- What optimizations can be used to make GPU Implementation faster?

But why Monte Carlo Simulation?

- Popular benchmark used in Computer Architecture field
- Does not need any special architectural features such as lookup tables etc.
- Need a measure of the throughput of computation

Overview of GPU

GPUs are compute intensive devices

- Each GPU has many Streaming Multiprocessors (SMs) which are lightweight cores
- Each SM runs one block of threads. If the num of blocks ¿ number of SMs, then, the blocks are re run on SMs
- The maximum number of threads per block depends on the architecture.

What does this mean to the programmer? The programmer is responsible for the synchronization! The programmer generates GPU kernels.

- There is a fixed maximum number of threads per block. If this is exceeded the kernel won't run.
- The number of threads per block is also limited.

CUDA GPU Kernels

How can a programmer handle all the exposed parallelism?

- A kernel is a function that is run by each thread in an SM.
- The scheduler in a GPU only schedules multiple blocks and launches threads in each block but is not responsible for thread synchronization
- The programmer is responsible for thread synchronization and safe thread execution.

Does it end here? NO!

- The programmer cannot assume the order of the execution of blocks or threads
- The programmer is also responsible to maintain safe execution across blocks!

GPU contd

Is is worth it?

- Yes! If the architecture is exposed a programmer has more flexibility
- Can explore various options such as varying number of threads, blocks, scheduling strategy
- Speedups reported are in the order of 10x-100x

Example of how a kernel is launched: Note:

- The parameters passed must be on GPU memory or registers. GPU cannot read CPU memory!
- The parameters passed are accessed by all threads in all blocks

Experiment Setup

Stampede Hardware

- CPU: Intel Xeon E-5
- GPU: NVIDIA K20

Software

- Languages used: C++ (CPU) , CUDA(GPU)
- Version Control: github
- Build System: make
- Plots: gnuplot
- Other scripts to automate and extract: bash, python

Implementation Details

The algorithm was divided into the following steps:

- Step 1: Generate a pair of random number from 0-1
  - CPU rand function used to generate 2*samples number of random numbers.
  - Values stored in an array and passed to both CPU and GPU estimation functions.
  - In GPU impl, the array is copied from CPU to GPU memory and passed to kernel
  - In the kernel, each thread with its unique thread index and block index accesses unique elements in the array

Implementation Details contd

- Step 2: Find the number of points within the quarter circle
  - A simple compute is used in both CPU and GPU impl to check whether a point lies within the quarter circle.
  - CPU: A count is maintained and incremented when the condition is true
  - GPU: If true, the thread writes 1 to its unique location in another shared array per block
  - : final count is calculated by adding all the 1s
- Step 3: Divide by total points to get the value of pi/4
  - CPU: Straightforward division of total values satisfying the condition by total samples
  - GPU: The final count obtained is per block. These per block sums have to be added to get final total count.
  - : Division by total samples is done outside the kernel by CPU because its a low cost computation

Optimizations

Naive implementation is slower. The programmer is responsible for juicing out the performance benefits from GPU. Here's how i did it:

- Calling rand function from CPU:
  - rand function API exists in CUDA but it is very slow and has high overhead
  - This is not fair to compare because the aim is to compare compute power and timing
  - Hence, CPU rand is used for both implementations

  Ping pong copy:
  - Input to perfix sum is the output of previous iteration.
  - This means each time the output global array has to be copied to input. Global memory -global memory copy is expensive in GPU.
  - Instead, alternate input and output arrays as output and input every odd and even iteration

Optimizations contd

- Why was 1 written to another memory instead of performing atomic adds?
  - Atomic adds are expensive and serialize the code
  - GPU being a highly parallel device handles massive serialization very badly increasing latencies.
  - A technique called Parallel Prefix Sum was used to calculate sum
- Optimizations in parallel prefix sum:
  - Loop unrolling: Standard optimization
  - Warp synchronization: Once less than 32 threads were forced to run, the warp scheduler scheduled and synchronized the threads.
  - This means expensive synchronize thread calls could be avoided

Previous Implementations

The following previous work (and open source) was found:

- CUDA Thurst Library has fast implementations of most popular simulations and algorithms
- cuRand Library also provided Monte Carlo pi estimation but the authors are still fixing compilation issues in their example code.
- Will be comparing my implementation with that of CUDA Thrust

Ensuring Fairness

Note on Thurst comparison:

- In Thurst implementation the very slow CUDA rand function has been used
- This makes it unfair to compare my implementation with Thurst
- Hence, another version was made which calls CUDA rand function.
- Still missing something: The time to allocate and free memory was also included. Now the comparison is fair.

Results: 1/3

Comparison with naive CPU implementation

- For the first few elements, CPU implementation is faster because the overhead of launching threads beats the benefits obtained from parallelism
- But for large number of samples, GPU implementation is approx 100x faster as shown below.

Results: 2/3

Comparison with CUDA Thrust Implementation

- My implementation is always faster than that of Thrust as shown below.
- From a glance at their code, potential for improvements were identified.

Results: 3/3

Other observations:

- Number of samples per thread for the fastest time is 1 or 2.
- I expected an increase in performance for 4, as a tradeoff between overhead of starting threads vs serializing the code by half.
- But, was midly surprised to find performance degradation

Directory Structure and Build System

- Free to use
- The following image describes the source file dependency

What would I have done differently?

- Satisfied with the general direction of the project from start
- Always scope for improvement
- Could have explored non open source fast implementations
- Could have modified thrust library, but implementing from scratch was much more fun!
- Could have given more flexibility to user
  - number of blocks, number of threads to be user defined
  - But will have to sacrifice warp scheduling

Conclusion

- Could exploit massive parallelism in GPU to beat Thrust implementation
- Good learning experience, confident as a multicore programmer!
- Could not have done without github
- Every researcher has a tendency to look for favourable results overlooking unfairness: Taught me a lesson to do fair comparisons

Example 1: 2 Blocks with nested item lists

First Block

- First Item
  - Subitem
- Second Item:
  - More subitems
  - And more

Second Block

- With an item

Example 2: 2 Columns;
one column with two blocks, one block with two columns!

Block 1

- item 1
- item 2
- item 3
- item 4
- item 5
- item 6

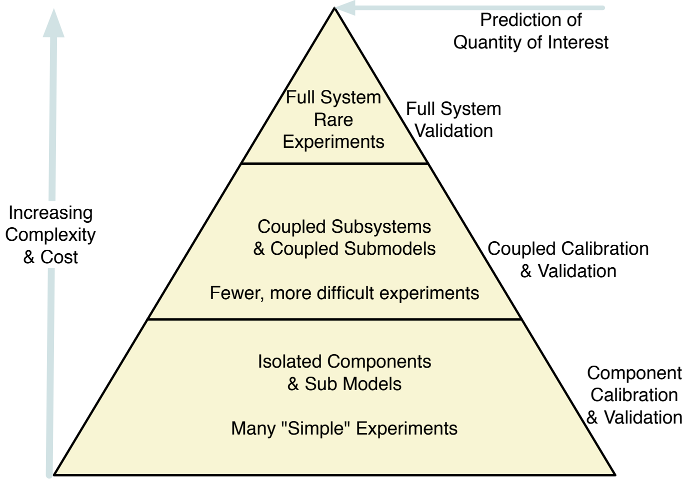
Block 2

- item A
- item B

Block 3

- item a
- item b
- item c
- item d

Image and Bullet points



- Validation is done repeatedly with increasingly complex scenarios
- Validation pyramid may be recursive

Two Blocks with added text for emphasis

V&V-UQ framework *requires* experimental data

Calibration of component model parameters

- Thermochemistry (e.g. kinetic parameters)
- Radiation (e.g. absorptions & emissions)
- Turbulence (e.g. model constants)
- Ablation (e.g. kinetic parameters)

Validation

- Component and subcomponent models
- Coupling between models
- Full system

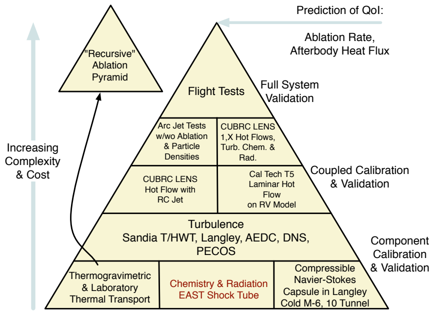
1 Block and 1 Image in Column format

Extensive experimental data

- Space Act Agreement
  - Ames EAST
  - Langley RCS
  - Ames & JSC Arc Jets
  - AEDC T9
  - CUBRC
  - Cal Tech
- Legacy data
- Sandia
- PECOS

Facility	Description	Flow	Measure	Calibration	Validate
UT	TGA	N/A	Mass(T) slow heat	Ablation Kinetics	Ablation
EAST	Shock Tube	Hypersonic	Radiometry	Chemistry, Radiation	Aerothermo, radiation
Langley	RV model	M=6,10,cold, laminar	q_s,T_s		Navier-Stokes
Langley RCS	RCS model	M=10,cold, laminar	q_S,P_s		Navier-Stokes
Sandia HWT	Sphere-cone model	M=5,8,14, cold	P_s,ρ_u	Turbulence	Turbulence
Sandia TWT	Turbulent BL w/steady cross-flow	M=0.8,cold	$u(2-D)$	Turbulence	Turbulence
Sandia TWT	Turbulent boundary layer	$M < 3$, cold	P_s, $u(2-D)$	Turbulence	Turbulence
Langley	Legacy Boundary layer experiments	$M < 11$, cold	ρ_u, T	Turbulence	Turbulence
AEDC T9	RV model w/wo roughness	M=6,cold	q_s, T_s	Turbulence	Turbulence, transition
ArcJet	PICA and copper targets	$M < 12$, hot,long	Particle density	Particles	Part. gen/ transport
ArcJet	Ablative material flow	$M < 12$, hot,long	q_s, T_s, σ_s, Recession		All
CUBRC LENS 1	Model w/ blowing / roughness	$M < 25$, hot	P_s, q_s, T_s, σ_s		All except ablation
CUBRC LENS	Model w/ RCS jets	$M < 25$, hot	P_s, q_s, T_s, σ_s		All except ablation
CUBRC LENS X	RV Model	$M < 25$, hot	P_s, q_s, T_s, σ_s		Turbulence, chemistry, radiation
CalTech T5	RV Model	$M < 5$, hot,laminar	q_s, T_s		Chemistry, radiation, transport
Fire II	Apollo-era flight test		q_s, T_s, Radiometry		All
Apollo IV	Apollo lunar exchange flight test		q_s, T_s, Radiometry		All
LEO	CEV Low exchange orbit		q_s, T_s, Radiometry		All
LEX	2m capsule lunar exchange		q_s, T_s, Radiometry		All
Stardust	Comet sample-return mission		TPS condition		All

1 image and 1 itemized Block



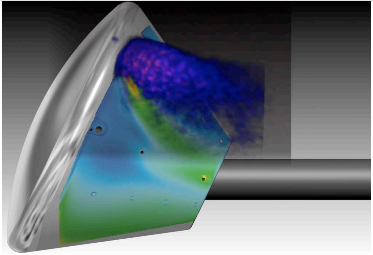
Goals

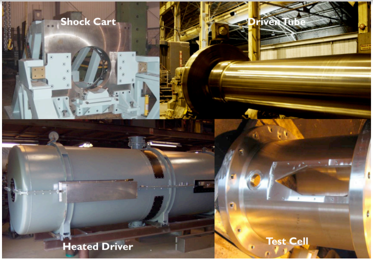
- Calibrate and (in)validate a two-temperature thermochemical model
- Investigate implementation of the validation cycle with QUESO
- Develop a 1D problem for future exploration of adjoints

Block and then Two Images in a Column

Facility

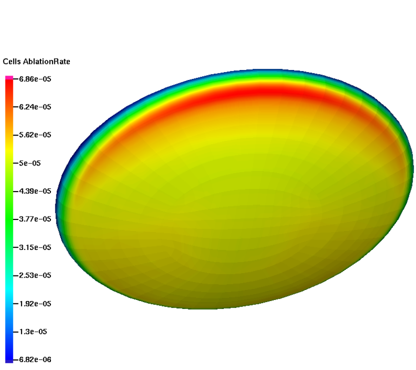
- LENS I HST
  - Variable Re reflected shock tunnel
  - Tests: Perfect gas data, enthalpy effects, distributed roughness, roughness w/ blowing, high-fidelity
- LENS XX
  - Variable Re shock expansion tunnel
  - Tests: Facility and measurement capabilities similar to EAST
- Visit planned for May 2009



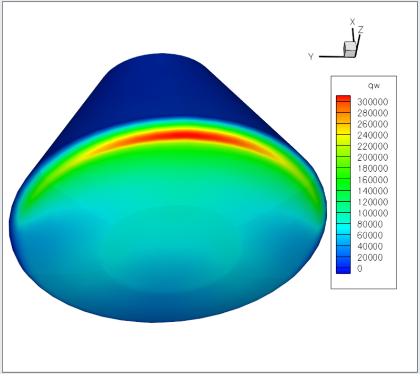


Two Images in Two Blocks in a Column

Surface Ablation Rate



Surface Heat Flux



Fancy Block / Column work

Goals

- Demonstrate capability to couple ablation and radiation models with existing hypersonic code (DPLR)
- Evaluate sensitivity of the ablation rate and peak heat flux (QoIs)
  - Identify most important models
  - Evaluate utility of surrogate quantities of interest

Coupled hypersonic flow for LEO and lunar reentry, including:

- Arrhenius chemistry
- Gray temperature dependent radiation
- Algebraic(Baldwin-Lomax) turbulence models
- Thermal nonequilibrium
- Single phase flow (i.e. no particles)

- 1-dimensional solid-phase ablation with ad hoc kinetics (as in CMA, FIAT, Chaleur)
- Equilibrium surface chemistry

Thank you!

Questions?