Parallelization of Monte-Carlo Estimation of pi

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 Build System
- 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 [2][3]
- 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.

CUDA GPU Kernels

How can a programmer handle all the exposed parallelism? Through kernels!

- 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

Note:

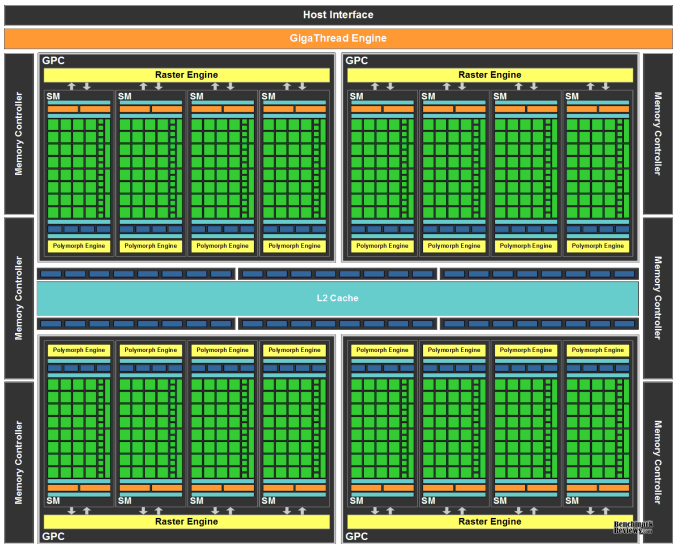
- The parameters passed must be on GPU memory or registers. GPU cannot read CPU memory!
- The parameters passed are accessed by all threads in all blocks

GPU Programming

How can GPUs be programmed?

- This project: in CUDA extended C++ supported by NVIDIA GPUs.
- OpenCL can be used for other GPUs

The picture below shows SMs on an NVIDIA GPU:



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

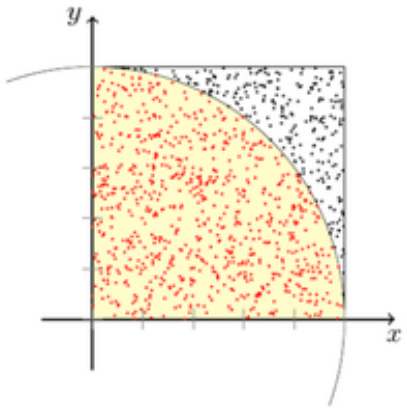
Implementation Details: 1/3

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: 2/3

- 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



Implementation Details: 3/3

- 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

Optimizations: 1/3

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

Optimizations: 2/3

- 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

```
int i=0
while(condition == true) {

   // call kernel
   prefix_sum_kernel<<<blocks,threads>>>
                    (d_in, d_result, length);
   // copy d_result to d_in
   cudaMemcpy(d_in, d_result,length*sizeof(int),
                     cudaMemcpyDevicetoDevice);
   // the above line is very expensive

   update condition ();
}
```

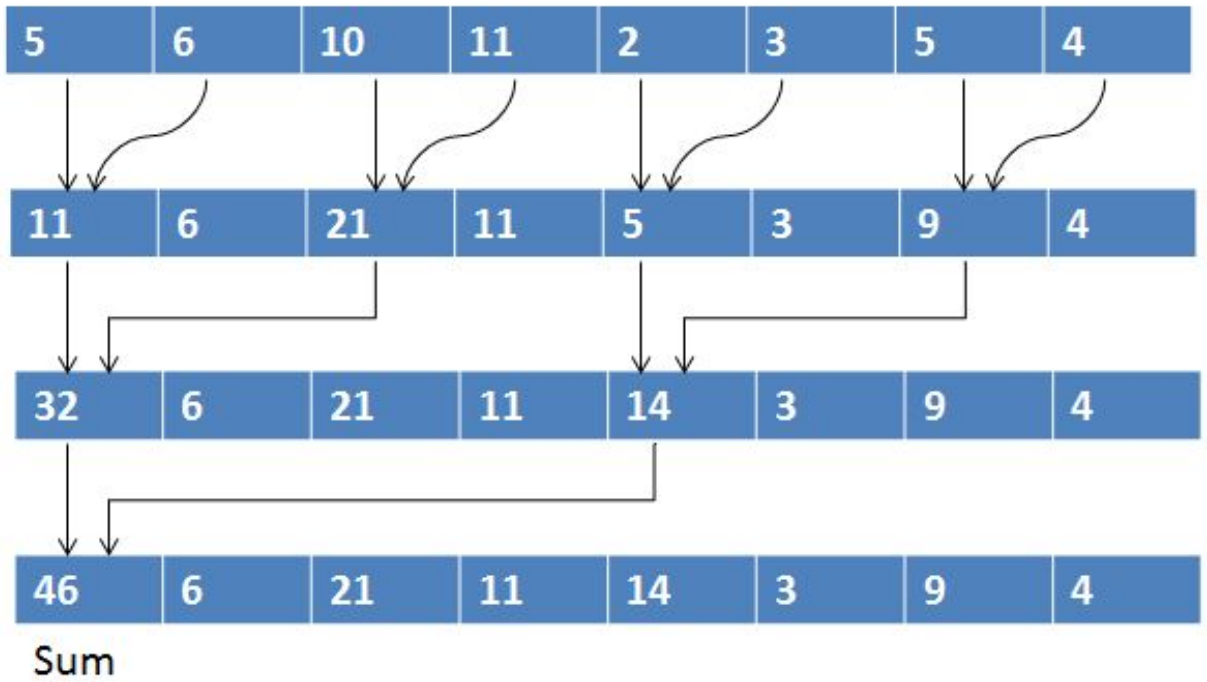
```
int i=0
while(condition == true) {

   if (i%2 == 0) {
       // call kernel
       prefix_sum_kernel<<<blocks,threads>>>
                        (d_in, d_result, length);
   } else {
       // call kernel
       prefix_sum_kernel<<<blocks,threads>>>
                        (d_result, d_in, length);
   }
    i++;
    update condition ();
}
```

Optimizations: 3/3

- 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

Parallel Prefix Sum Example

The arrows clearly indicate the potential to parallelize the algorithm



Previous Implementations

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

- CUDA Thurst Library has fast implementations of most popular simulations and algorithms [1]
- cuRand Library also provided Monte Carlo pi estimation but the authors are still fixing compilation issues in their example code [4]
- Will be comparing my implementation with that of CUDA Thrust

Ensuring Fairness

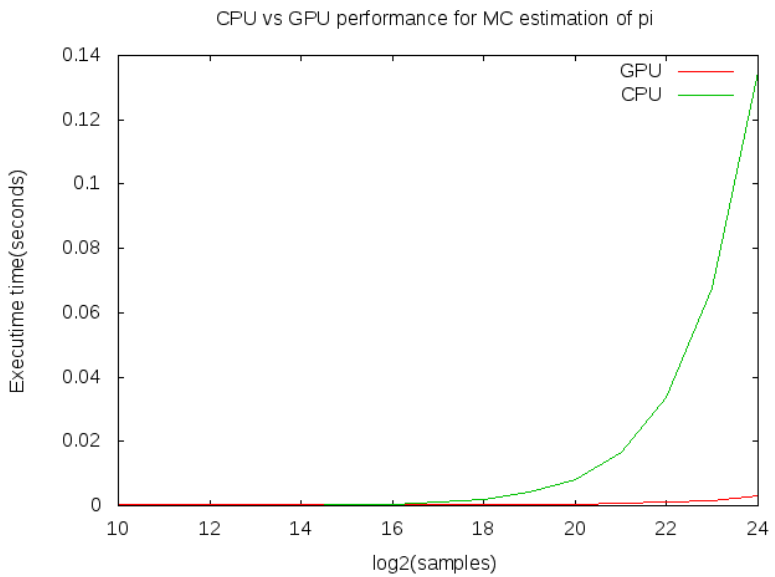
Note on comparison with Thrust

- 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

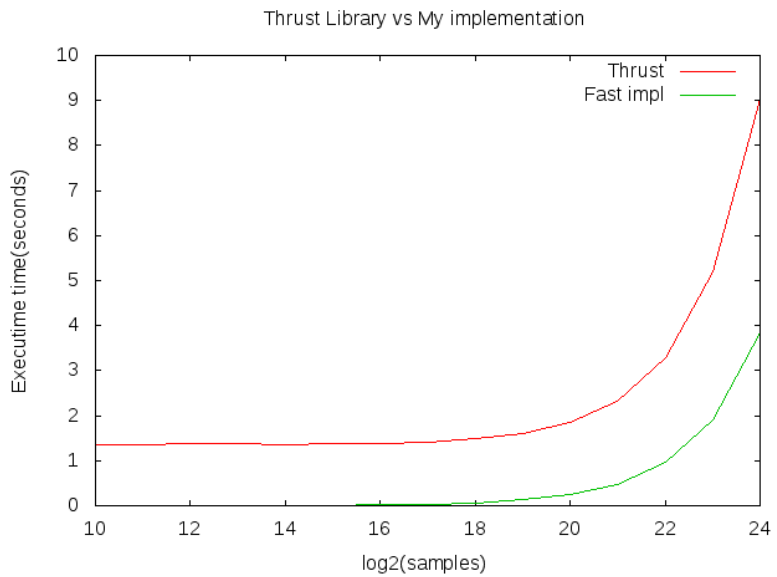
- For the first few elements, CPU implementation is faster because the overhead of launching threads beats the benefits obtained from parallelism
- But for large number of samples, GPU implementation is approx 100x faster as shown below.



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.



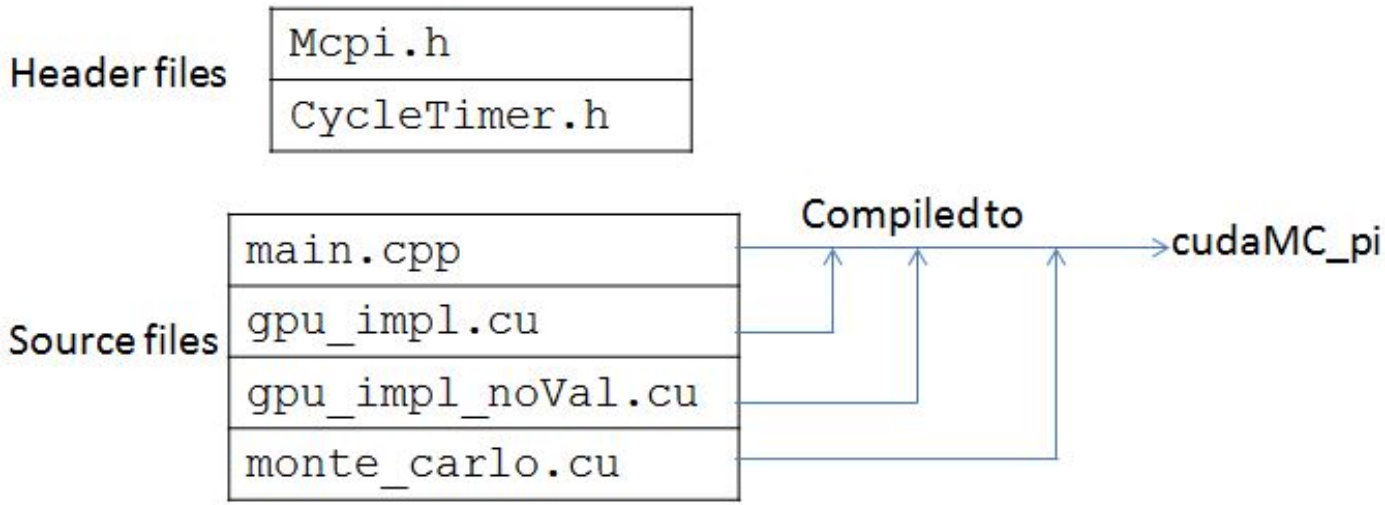
Results: 3/3

Other Observations

- Number of samples per thread for the fastest time is 1 or 2.
- I expected an increase in performance for 4: 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: public repository on github
- The following image describes the source file dependency



What would I have done differently?

- Satisfied with the general direction of the project from start
- There's 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

References

[1] www.docs.nvidia.com/cuda/thrust
[2] Lee et al, Debuking the 100x GPU vs CPU myth: an evaluation of throughput computing on CPU and GPU. ACM SIGARCH Computer Architecture News 38.3 2010.
[3] Thomas et al, A comparison of CPUs, GPUs, FPGAs, and massively parallel arrays for random number generation. Proceedings of ACM SIGDA Internation Symposium on FPGAs, ACM 2009.
[4] www.docs.nvidia.com/cuda/curand

Thank you!

Questions?