Karlsruher Institut für Technologie

Monte Carlo Method on GPUs

Topic 05

Roland Osterrieter

STEINBUCH CENTRE FOR COMPUTING - SCC

Agenda

- Motivation
- Monte Carlo Method
- Nvidia CUDA
- Program for π calculation
- Running on ForHLR 2
- Results
- Conclusion
- Sources

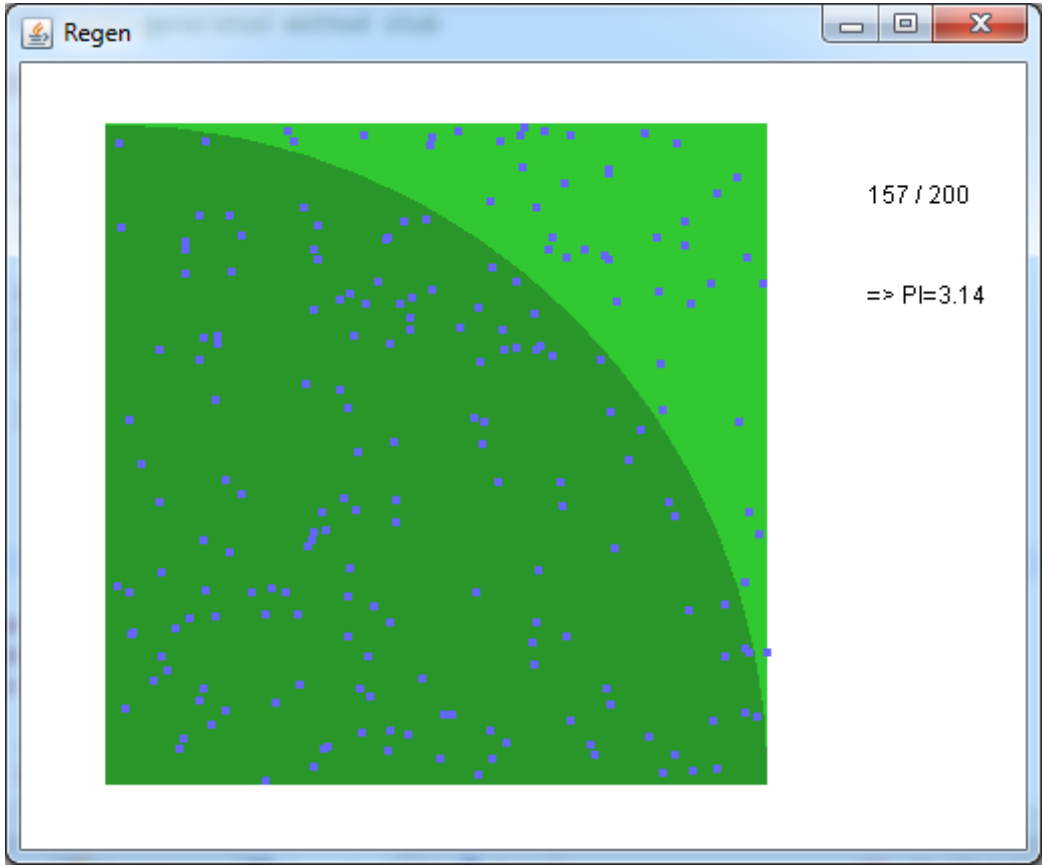
Motivation

- Implement a monte carlo algorithmus for GPU
- Run in a GPU cluster for huge problem sizes
- Compare with a „standard“ cpu implementation

Monte Carlo Method

- => Use the law of large numbers
- Huge sizes of random samples
- Approximate a result

- Famous problem: π calculation
  - „Raindrops in a grass circle“
  - $A_C = \pi r^2 \;\; A_S = a^2 = (2 * r)^2$
  - $\frac{A_c}{A_S} = \frac{\pi r^2}{(2*r)^2} = \frac{\pi}{4}$
  - $\pi = \frac{Drops\ in\ Circle}{Number\ Of\ Drops} * 4$



Nvidia CUDA

- Platform that enables general purpose calculations on GPUs
- GPU is specialized for compute-intensive, highly parallel computation
- Programming model:
  - Kernel code that gets executed on the device
  - Data transfer between host and device
  - Kernels run in blocks, which contain threads

Program for π calculation

- Written in C, compiled with CUDA nvcc
- Using curand library for calculating random numbers on the device

- Kernel Procedure:
  - Blocks use shared memory to store the counters for their threads
  - All threads run a for-loop with the drop-calculation logic
  - One thread in each block aggregates the counters of all threads

- Post Processing:
  - The aggregated counter of each block gets copied to the host
  - The host measures the execution time and calculates pi with the summed up counters

Program for π calculation

- Drop-calculation loop

```
curandState_t curandom;
curand_init(clock64(), uniqId, 0, &curandom);
threadTotals[threadId] = 0;
for (Counter i = 0; i < ITERATIONS; i++)
{
    float x = curand_uniform(&curandom) ;
    float y = curand_uniform(&curandom) ;
    float distanceToCenter = sqrt(x*x + y*y);
    bool inCircle = distanceToCenter <= 1;
    if (inCircle) {
        threadTotals[threadId]+=1;
    }
}
```

Running on ForHLR 2

- Rendering-nodes with 48 cores (4xNvidia GeForce GTX980 Ti)
- CUDA Toolkit version 9.0
- Access a node with remote client

- Running the kernel:
  - Threads per block: Set to warpsize (32)
  - Drop-calculation loop set to 10.000.000 iterations
  - Tested with different blockcounts to gather data
  - Total drops = Blockcount * 32 * 10.000.000

Sample Output

- Run on node with 256 blocks

WARP SIZE: 32
BLOCKS: 256
ITERATIONS: 10000000
Init
Call Kernel
Aggregate Results
Finished calculating in 1.642886 seconds!
=> 64340018266 in Circle of
=> 81920000000 Points
=> PI = 3.141602454394531118 71
[PI is = 3.14159265358979323846]
(1.6 Seconds)

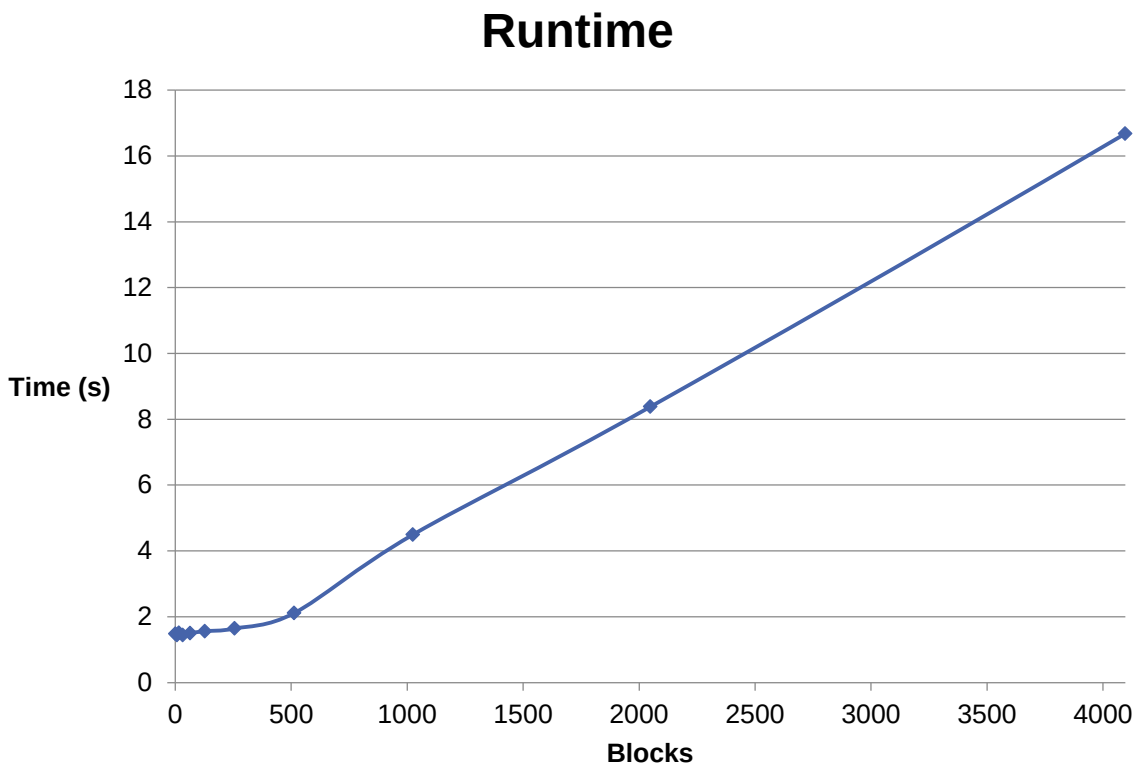
Comparison:

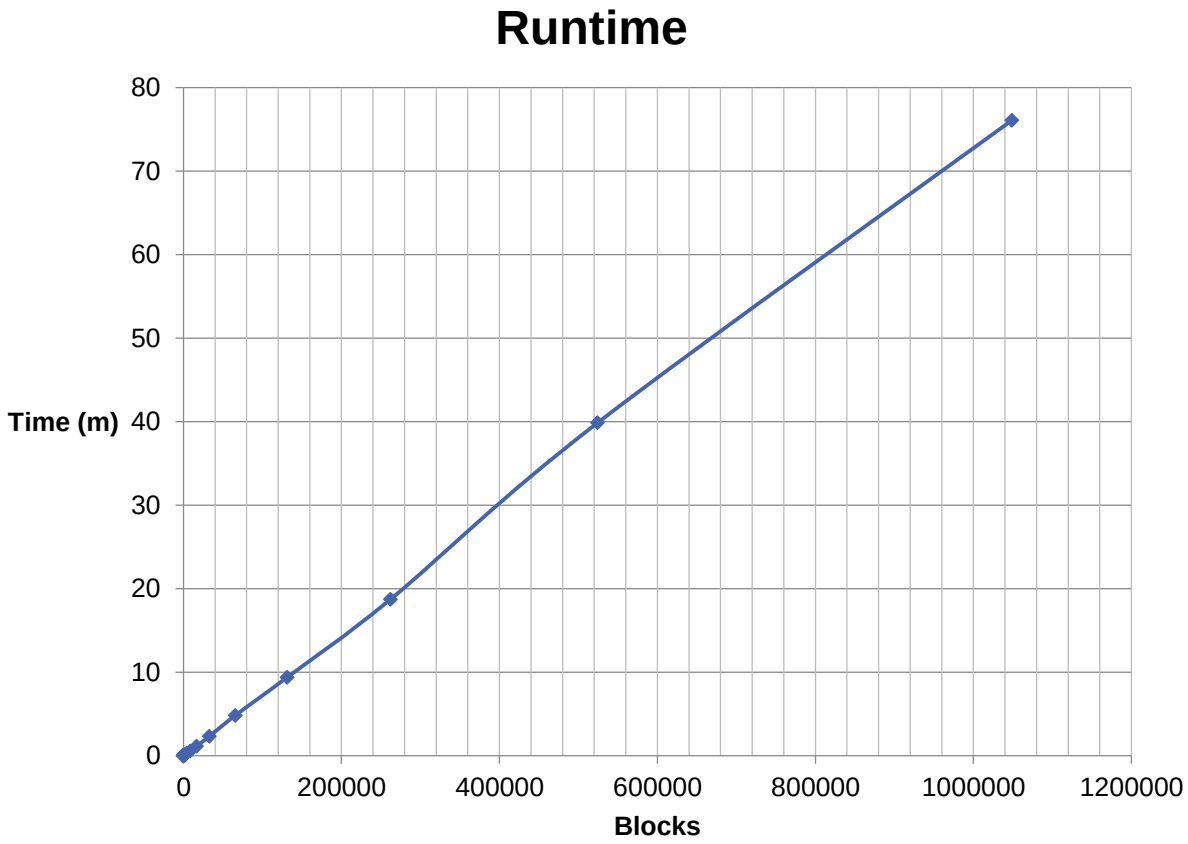
- Simple Java implementation
- For-loop, no multithreading

Finished calculating in 2257.726 Seconds!
=> 64339653084 in Circle of
=> 81920000000 Points
=> PI = 3.141584623242 1876
[PI is = 3.14159265358979323846]

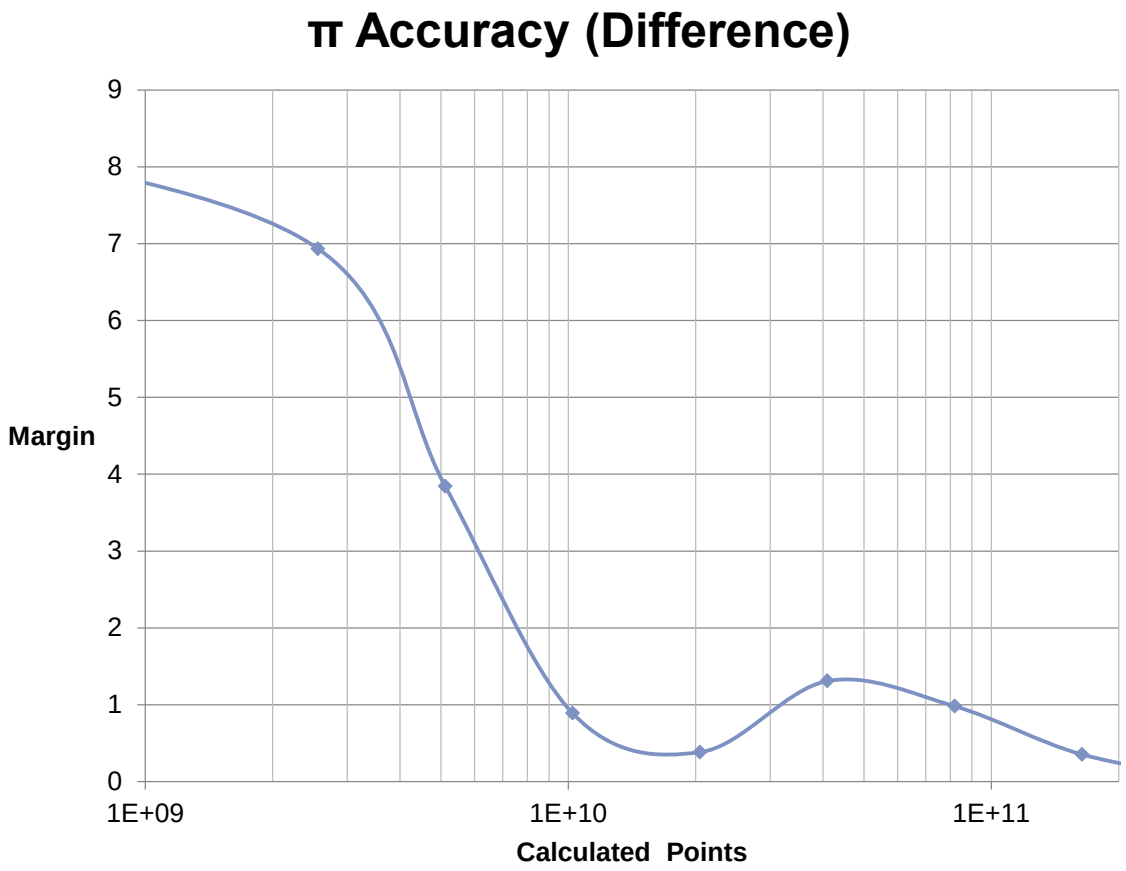
(38 Minutes)

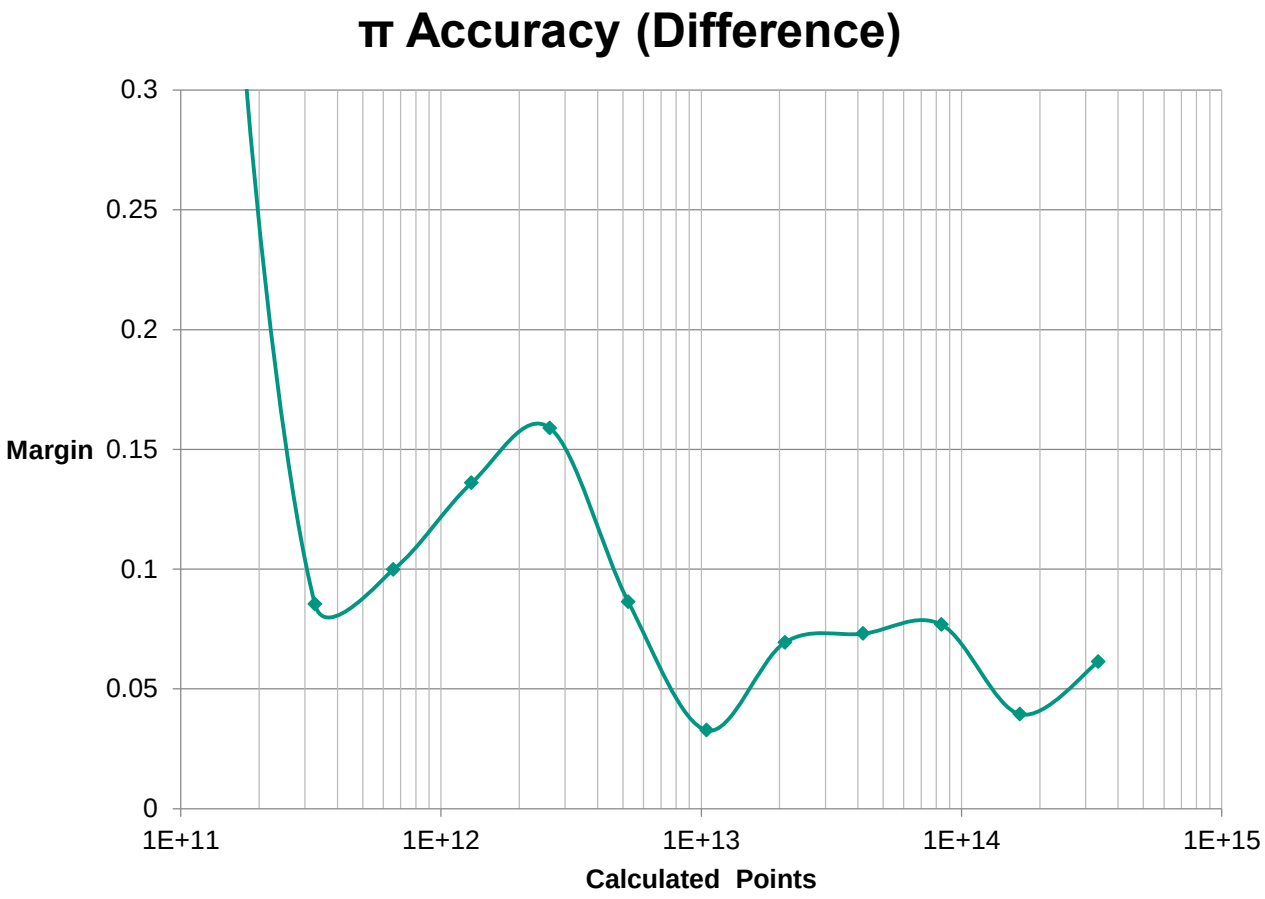
Results - Runtime





Results – π Accuracy





$$Margin = ABS(Calc.\pi - \pi) * 100000$$

Conclusion

- Accuracy not strictly improving with bigger runs
  - Randomness and quality of the random number generator
- Trend is showing a slow improvement overall
  - Good results with small drop counts already
  - $\pi = 3.14$ with ~200 drops
  - $\pi = 3.14519$ with ~10.000.000.000 drops
  - Best result had 6 correct digits ($\pi = 3.145192$)
- Monte Carlo method useful for other problems
  - Not the „best“ showcase, but a simple one
  - Better for problems that are hard or impossible to solve deterministic

Sources

- Source code:

https://github.com/Rolleander/gpuPi

- CUDA logo:

https://nvidianews.nvidia.com/file?fid=544a60fef6091d588d000046

- CUDA guide:

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction

- CUDA best practices for c:

http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf

- SCC ForHLR2 picture / website:

https://www.scc.kit.edu/dienste/forhlr2.php

Sources – Results Table

blocks	points	time (s)	pi calc	pi margin
1	320000000	1.480765	3.14150791	8.47410898
8	2560000000	1.440004	3.141662	6.93448477
16	5120000000	1.507164	3.14163106	3.841047271
32	10240000000	1.445042	3.14158378	0.887351167
64	20480000000	1.500662	3.14158883	0.381862886
128	40960000000	1.561434	3.14157958	1.307751558
256	81920000000	1.642886	3.14160245	0.980080474
512	1.6384E+11	2.109586	3.14159618	0.352575591
1024	3.2768E+11	4.491677	3.1415918	0.085367524
2048	6.5536E+11	8.377587	3.14159365	0.099825347
4096	1.31072E+12	16.675217	3.14159129	0.136008394
8192	2.62144E+12	33.776812	3.14159424	0.158932555
16384	5.24288E+12	68.858016	3.14159352	0.086350081
32768	1.04858E+13	139.506641	3.14159233	0.032703492
65536	2.09715E+13	289.252594	3.14159335	0.069416564
131072	4.1943E+13	562.791313	3.14159338	0.073036311
262144	8.38861E+13	1124.12813	3.14159342	0.076787241
524288	1.67772E+14	2392.2615	3.14159305	0.039411588
1048576	3.35544E+14	4564.098	3.14159327	0.061338492

$$\pi = 3.14159265358979$$
$$Margin = ABS(Calc.\pi - \pi) * 100000$$