GPGPU Monte Carlo

CSC 490: Lock-free, GPU & vectorization

Assignment 3

Victor Kamel
vkamel@uvic.ca
University of Victoria
Victoria, BC, Canada

1 INTRODUCTION

In this report, we will calculate the value of the mathematical constant π using the Monte Carlo simulation method. We will compare the performance of algorithms running on both the CPU and the GPU.

2 PROBLEM STATEMENT

The Monte Carlo simulation method involves iteratively converging to a quantity by randomly sampling from a statistical distribution.

Suppose that we have $Z = (X, Y)$ with $X, Y \overset{iid}{\sim} U(-1, 1)$. Now, we take the probability $\mathbb{P}(Z \in A) = p$ with $A = \{(x, y) \mid x^2 + y^2 \leq 1\}$. That is to say the probability of sampling a point within the unit circle centered at the origin.

$$p = \mathbb{P}(X^2 + Y^2 \leq 1) = \int_{-1}^{1} \int_{-\sqrt{1-y^2}}^{\sqrt{1-y^2}} p_{X,Y}(x, y)\, \mathrm{d}x\, \mathrm{d}y$$

$$p_{X,Y} = \frac{1}{1-(-1)} \cdot \frac{1}{1-(-1)} \qquad (\text{uniform}, \perp\!\!\!\perp)$$

$$p = \int_{-1}^{1} \int_{-\sqrt{1-y^2}}^{\sqrt{1-y^2}} \frac{1}{4}\, \mathrm{d}x\, \mathrm{d}y = \frac{\pi}{4}$$

$$\pi = 4p$$

We will estimate p by the proportion of uniform random samples that are within the unit circle as illustrated in Figure 1. As the number of samples grows, our confidence in the precision of π grows. Without loss of generality, we can also use only the upper right quadrant along with $U(0, 1)$.

3 ALGORITHMS

Both the quality and speed of the random number generation will be important. We will be using pseudo-random number generators (PRNGs) rather than true random number generators (TRNGs), as the latter would be too slow.

3.1 CPU

We use the Mersenne Twister (`std::mt19937` in the C++ standard library) seeded with `std::random_device`. Each thread will randomly seed its own PRNG since the samples between threads must be independent. Each thread will generate a certain number of samples, and record the count that fall within the circle. Once the threads have completed, the partial counts from each thread are summed sequentially.

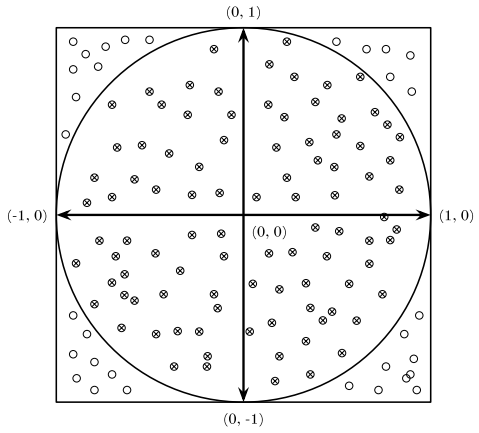

Figure 1: Calculating π by sampling a uniform distribution.

```cpp
std::pair<unsigned, unsigned> calculate_samples_cpu()
{
    const std::size_t n_threads = std::thread::hardware_concurrency();
    const std::size_t samples_per_thread = ceil(MT_TARGET_SAMPLES / n_threads);
    const std::size_t n_samples = samples_per_thread * n_threads;

    std::vector<std::size_t> samples(n_threads);

    {   // Thread pool
        std::vector<std::jthread> spool;
        spool.reserve(n_threads);

        // Start threads
        for (std::size_t n = 0; n < n_threads; ++n) {
            spool.emplace_back([&samples, samples_per_thread, n] {
                std::random_device rd;
                std::mt19937 gen(rd());
                std::uniform_real_distribution<float> dist(-1.0f, 1.0f);

                std::size_t count = 0;
                for (std::size_t i = 0; i < samples_per_thread; ++i) {
                    float x = dist(gen);
                    float y = dist(gen);
                    count += (x * x) + (y * y) <= 1.0f;
                }
                samples[n] = count;
            });
        }
    }

    return std::pair<unsigned, unsigned>(std::accumulate(samples.cbegin(), samples.cend(), 0), n_samples);
}
```

3.2 GPU

On the GPU, we use a hybrid PRNG based on the Tausworth PRNG with three parameters and a Linear Congruent Generator with one parameter, as presented in Chapter 37 of [2]. According to the authors, this PRNG has a period of around 2^{121}. This is lower then the Mersenne Twister's period of $2^{19,937}$, but should still be acceptable.

```
// From GPU Gems 3 Ch. 37 (Lee Howes and David B. Thomas)
__device__ unsigned TausStep(unsigned &z, int S1, int S2, int S3,
    ↪ unsigned M)
{
    unsigned b = (((z << S1) ^ z) >> S2);
    return z = (((z & M) << S3) ^ b);
}

// From GPU Gems 3 Ch. 37 (Lee Howes and David B. Thomas)
__device__ unsigned LCGStep(unsigned &z)
{
    return z = (1664525 * z + 1013904223);
}

// From GPU Gems 3 Ch. 37 (Lee Howes and David B. Thomas)
__device__ float getRandomValueTauswortheUniform(unsigned &z1,
    ↪ unsigned &z2,
                                                 unsigned &z3,
    ↪ unsigned &z4)
{
    unsigned taus = TausStep(z1, 13, 19, 12, 4294967294UL)
                  ^ TausStep(z2, 2, 25, 4, 4294967288UL)
                  ^ TausStep(z3, 3, 11, 17, 4294967280UL);
    unsigned lcg = LCGStep(z4);

    return 2.3283064365387e-10f * (taus ^ lcg); // taus + lcg
}
```

In order to seed the 4 parameters of the PRNG for each thread, we use the Mersenne Twister on the CPU.[1] This ensures that that the quality of the seeds is high. The contiguous array of seeds is then copied over to the GPU in global memory. Thus, for each thread, the kernel picks up its parameters from the given input. Each thread generates a certain number of pairs of random floats, and counts the number whose coordinates are within the circle.

```
__global__ void runSimulation(unsigned *seedPool)
{
    unsigned z1, z2, z3, z4;
    unsigned addr = (blockIdx.x * blockDim.x + threadIdx.x) *
      ↪ TW_NSEEDS_PER_RNG;
    z1 = seedPool[addr    ]; z2 = seedPool[addr + 1];
    z3 = seedPool[addr + 2]; z4 = seedPool[addr + 3];

    // Note: We will not save RNG state for future invocations

    float x, y;
    unsigned count = 0;
    for (unsigned loop = 0; loop < TW_SAMPLES_PER_THREAD; ++loop)
    {
        x = getRandomValueTauswortheUniform(z1, z2, z3, z4);
        y = getRandomValueTauswortheUniform(z1, z2, z3, z4);
        count += (x * x) + (y * y) <= 1.0f;
    }

    reduceInto(&d_result, count);
}
```

Once the samples have been processed, we could write the PRNG parameters back to global memory for a future invocation. We will omit this here.

[1]Note that the seeds must have a value of at least 128 [2].

Finally, the thread counts must be reduced to a single count. Our original implementation copied the randomly generated numbers to the CPU directly, and the simulation was performed there. However, this is slow and inefficient. Thus, we adopted a parallel approach with shared memory. This is done in a block-wise fashion, according to the method described in [1]. Each thread writes its count into shared memory, then the threads are synchronized within each block. Next, the values of the threads are added in a pairwise fashion, synchronizing at each step. Finally, one thread per block writes the bock sum to the output variable atomically.

```
// Block-wise shared memory parallel reduce (Meister et al. 2023)
template <class T>
__device__ void reduceInto(T* loc, T val)
{
    __shared__ unsigned smem[TW_NTHREADS_PER_BLOCK];
    smem[threadIdx.x] = val;

    __syncthreads(); // Synchronize

    for (int i = 1; i < blockDim.x; i <<= 1)
    {
        if (threadIdx.x < (threadIdx.x ^ i))
            { smem[threadIdx.x] += smem[threadIdx.x ^ i]; }
        __syncthreads(); // Synchronize
    }

  if (threadIdx.x == 0) { atomicAdd(loc, smem[0]); } // Coalesce
}
```

4 PERFORMANCE EXPERIMENTS

	Execution Time	Mean Squared Error
CPU	19189.8 ms	1.728×10^{-8}
GPU	125.8 ms	1.914×10^{-9}

Table 1: Performance comparison (5 run average).

Performance measurements were taken on the Google Colab cloud platform, with the Nvidia Tesla T4 GPU. However, the CPU allocation only allows 2 threads. Thus, the CPU algorithm is more limited then it would otherwise be. We chose to test both algorithms on the same platform despite this limitation, since the GPU algorithm also has a CPU component (seeding the PRNGs). The full execution time is measured, including memory copies from the host to device and vice versa. [2] In addition, the initial run of the GPU code was slower (likely due to the cache or initializing the device), and was thus discarded. The GPU kernel was run with 64 blocks and 1024 threads per block, based on the characteristics of the device and experiments. A total of 2^{28} samples were generated since at least 1×10^8 samples were observed to be required to produce three decimal places consistently, and this allows each thread to generate 2^{12} samples in the GPU, and 2^{27} samples in the CPU.

From Table 1, we can see that the GPU algorithm has a 152.5 times improvement on the CPU algorithm, as expected. We note that the MSE is also lower by a factor of 10, which is counterintuitive since we would expect that the Mersenne Twister to be a better source of randomness. Since these quantities are random, we submit that 5 samples is insufficient to characterize the MSE.

[2]Note that the profiling done in the Figure 2 demonstrates the runtime without data transfers/allocation.

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   98.51%  5.9493ms         1  5.9493ms  5.9493ms  5.9493ms  runSimulation(unsigned int*)
                    1.45%  87.646us         1  87.646us  87.646us  87.646us  [CUDA memcpy HtoD]
                    0.04%  2.1760us         1  2.1760us  2.1760us  2.1760us  [CUDA memcpy DtoH]
      API calls:   50.59%  132.47ms         1  132.47ms  132.47ms  132.47ms  cudaMalloc
                   46.91%  122.83ms         1  122.83ms  122.83ms  122.83ms  cudaLaunchKernel
                    2.28%  5.9589ms         1  5.9589ms  5.9589ms  5.9589ms  cudaMemcpyFromSymbol
                    0.10%  267.66us         1  267.66us  267.66us  267.66us  cudaMemcpy
                    0.06%  148.72us         1  148.72us  148.72us  148.72us  cudaFree
                    0.05%  142.46us       114  1.2490us     135ns  53.408us  cuDeviceGetAttribute
                    0.00%  10.801us         1  10.801us  10.801us  10.801us  cuDeviceGetName
                    0.00%  7.5030us         1  7.5030us  7.5030us  7.5030us  cuDeviceGetPCIBusId
                    0.00%  3.4630us         1  3.4630us  3.4630us  3.4630us  cuDeviceTotalMem
                    0.00%  1.8320us         3     610ns     211ns  1.3680us  cuDeviceGetCount
                    0.00%  1.0990us         1  1.0990us  1.0990us  1.0990us  cudaGetLastError
                    0.00%  1.0670us         2     533ns     205ns     862ns  cuDeviceGet
                    0.00%     704ns         1     704ns     704ns     704ns  cuDeviceGetUuid
                    0.00%     524ns         1     524ns     524ns     524ns  cuModuleGetLoadingMode
```

Figure 2: `nvprof` performance report.

```
------------------------------ ----------- ------------
Metric Name                    Metric Unit Metric Value
------------------------------ ----------- ------------
Block Limit SM                       block           16
Block Limit Registers                block            2
Block Limit Shared Mem               block            8
Block Limit Warps                    block            1
Theoretical Active Warps per SM       warp           32
Theoretical Occupancy                    %          100
Achieved Occupancy                       %        99.99
Achieved Active Warps Per SM          warp        32.00
------------------------------ ----------- ------------
```

Figure 3: `ncu` occupancy report.

4.1 Profiling

In addition to measuring the performance of the complete application, we profiled the GPU code with the `nvprof` and `ncu` command line utilities. As seen in Figure 2, most of the time spent by the GPU is spent in the `runSimulation` function. In addition, this duration is significantly shorter than the time required to allocate the memory and to launch the kernel. Thus, the number of random samples generated per core can be greatly increased with a relatively small increase in runtime. Such scaling characteristics are much better than that of the CPU algorithm. Note that from this performance report, we can determine that this is also much better than using the original approach without shared memory, since we can see that time to copy a single integer value from the GPU to the CPU is about the same as the runtime of the kernel. The additional overhead of copying thread counts back to the CPU would thus be very large (and using global memory instead of shared memory for the reduction would also be much slower). In addition, Figure 3 demonstrates that we are near the theoretical limit for occupancy.

5 CONCLUSION

We have implemented both a CPU and a GPU algorithm for calculating the value of π via the Monte Carlo, as well as demonstrated the advantage of the latter, providing a 152.5 times speed improvement. Further research directions involve better characterizing the error in both approaches and calculating the theoretical number of samples required to attain a given precision with a statistically high degree of confidence.

REFERENCES

[1] Daniel Meister, Atsushi Yoshimura, and Chih-Chen Kao. 2023. GPU Programming Primitives for Computer Graphics. In *ACM SIGGRAPH Asia 2023 Courses* (Sydney, Australia) *(SIGGRAPH Asia 2023)*. ACM, New York, NY, USA.
[2] Hubert Nguyen. 2007. *GPU Gems 3.* Addison-Wesley Professional.