Project Report: Parallel Implementation of the Bootstrap Particle Filter Using αSMC on CUDA/GPU

Ertuğ Karamatlı

January 18, 2014

1 Introduction

The αSMC algorithm [1] is proposed recently. The key idea is a formal generalization of the resampling step common to various sequential Monte Carlo algorithms.

Consider the HMM

$$x_0 \sim \mu_0 \tag{1}$$
$$x_n|x_{n-1} \sim f(x_n|x_{n-1}), n \geq 1 \tag{2}$$
$$y_n|x_n \sim g(y_n|x_n), n \geq 0 \tag{3}$$
$$\tag{4}$$

We let $g(y_n|x_n) \equiv g_n(x_n)$. The inferential goal is the computation of the prediction densities

$$p(x_n|y_{0:n-1}) \equiv \quad \pi_n(x_n) = \frac{Z_{n-1}}{Z_n} \int dx_{n-1} f(x_n|x_{n-1}) g_{n-1}(x_{n-1}) \pi_{n-1}(x_{n-1}) \tag{5}$$

and the associated normalization constants

$$Z_n = Z_{n-1} \int dx_{n-1} g_{n-1}(x_{n-1}) \pi_{n-1}(x_{n-1}) \tag{6}$$

for $n = 0$ **do**
  for $i = 1 \dots N$ **do**
    Set $W_n^i \leftarrow 1$
    Sample $\zeta_n^i \sim \mu_0$
  end for
end for
for $n = 1 \dots$ **do**
  Select α_{n-1} according to $\{\zeta_0, \dots, \zeta_{n-1}\}$
  for $i = 1 \dots N$ **do**
    Set

$$W_n^i \leftarrow \sum_j \alpha_{n-1}^{ij} W_{n-1}^j g_{n-1}(\zeta_{n-1}^j)$$

Sample

$$\zeta_n^i \sim \frac{1}{W_n^i} \sum_j \alpha_{n-1}^{ij} W_{n-1}^j g_{n-1}(\zeta_{n-1}^j) f(x_n|\zeta_{n-1}^j)$$

end for
end for

The filtering density and the likelihood are approximated by

$$Z_n^N = \frac{1}{N} \sum_i W_n^i \tag{7}$$

$$\pi_n^N(x_n) = \frac{\sum_i W_n^i \delta(x_n - \zeta_n^i)}{\sum_i W_n^i} \tag{8}$$

2 Example problem

We define the following example dynamical system

$$x_t|x_{t-1} \sim f(x_t|x_{t-1}) = \mathcal{N}(x_t; x_{t-1}, S) \tag{9}$$

Instead of defining observation we define sequence potential functions that have multimodality

$$g_t(x_t) = \sum_i^K a_i(t) \mathcal{N}(x_t; \mu_i(t), P_i) \tag{10}$$

3 CUDA

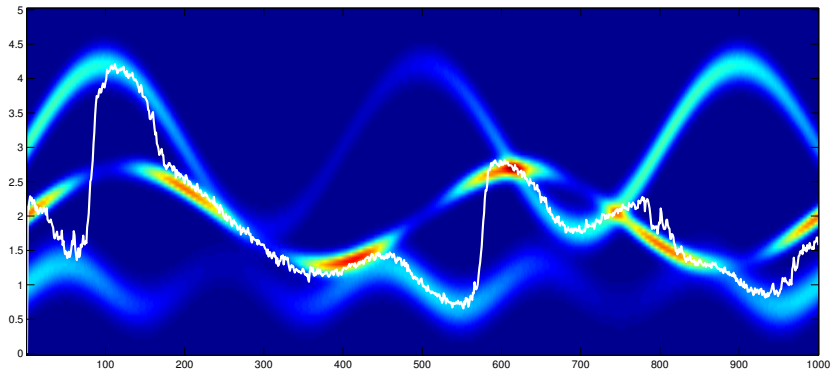
In CUDA [2], computation can be done in both CPU and GPU, namely, called host and device, respectively. CUDA C programs start executing on the host and it can launch kernels to run code on the device. Kernels are specially defined C functions that executes in parallel for each thread on the device.

Some of the important concepts in CUDA are threads, blocks, and grids. Blocks are a group of threads that has a shared memory. Blocks can access their own shared memory and global memory but cannot access another block's shared memory. A grid is a group of blocks. Blocks in a grid can only communicate via the global memory.

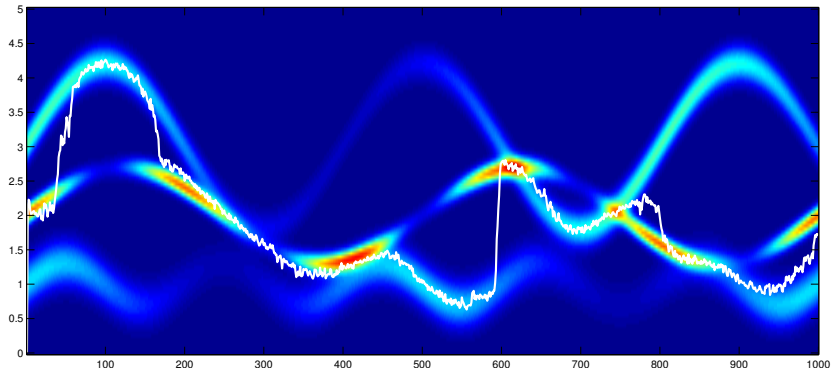
4 Implementation

We have implemented the bootstrap particle filter variant of the αSMC algorithm using CUDA. The implementation uses a single block of threads which allows efficient shared memory accesses. On the other hand, using a single block limits the simulation to have a maximum of 256 to 1024 particles, depending on the device used.

To have an unlimited number of particles we need to use multiple blocks that shares state using global memory. Also, synchronization of multiple blocks requires multiple kernel launches thus adding a small overhead.



(a) MATLAB



(b) CUDA

Figure 1: Comparison of the two implementations for validation. The white lines show the MMSE estimate. $N = 256$, $T = 1000$

4.1 Pseudorandom Number Generation on the GPU

CURAND library of the CUDA toolkit allows generation of pseudorandom and quasirandom numbers on the CPU or on the GPU. In order to generate pseudorandom numbers in parallel and without communication overhead, CURAND provides a way to start each thread at a different point in the sequence (skip-ahead). CURAND contains functions to sample from uniform, normal, log normal, and Poisson distributions efficiently.

4.2 Parallel Reduction

Parallel implementation of array operations such as summing up all values or finding the maximum value is challenging because writes to a single memory address should be serialized. In large arrays this serialization is significant because only one of the many cores of the GPU can be utilized.

The standard method for improving utilization of cores is parallel reduction [3] which allows only pairwise operations. Computation is done in a tree form that takes $\mathrm{O}(\log N)$ steps.

References

[1] N. Whiteley, A. Lee, and K. Heine, "On the role of interaction in sequential Monte Carlo algorithms," *ArXiv e-prints*, Sept. 2013.

[2] "CUDA C Programming Guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[3] M. Harris, "Optimizing Parallel Reduction in CUDA." `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`.

Appendix: CUDA Code

```
#include <stdio.h>
#include <curand_kernel.h>

#define T 1000
#define N 256

__device__ int discrete_rand(curandState *state, float *p)
{
    float u = curand_uniform(state);
    float z = 0;
    for (int i = 0; i < N; i++) {
        z += p[i];
        if (u < z) {
            return i;
        }
    }
    return 0;
}

__device__ float normal_pdf(float x, float mu, float sigma)
{
    return 1/(sigma*sqrt(2*M_PI))*exp(-pow(x-mu,2)/(2*pow(sigma,2)));
}

__device__ float reduce_sum(int idx, float value)
{
    __shared__ float arr[N];
    arr[idx] = value;
    __syncthreads();

    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (idx < s) {
            arr[idx] += arr[idx + s];
        }
        __syncthreads();
    }
```

```
    return arr[0];
}

__device__ float eval_f(curandState *state, float x)
{
    float S = 0.2;
    return x + sqrt(S)*curand_normal(state);
}

__device__ float eval_g(float x, int t)
{
    float K = 3;
    float P[3] = {0.05, 0.01, 0.03};
    float Tmu[3] = {200, 500, 400};
    float Tw[3] = {500, 200, 800};
    float A[3] = {0.3, 0.7, 1.2};

    float g = 0;
    for (int i = 0; i < K; i++) {
        float mu = (i+1)+A[i]*sinf(2*M_PI*t/Tmu[i]);
        float w = 1.01+0.8*cosf(2*M_PI*t/Tw[i]);
        g += w*normal_pdf(x, mu, sqrt(P[i]));
    }
    return g;
}

__global__ void asmc_bpf(curandState *state, float *E_d)
{
    int idx = threadIdx.x;

    __shared__ float xx[N];
    __shared__ float xx_prev[N];
    __shared__ float p[N];

    curandState localState = state[idx];
    float g;
    float xx_idx;
    float g_sum;
    float xx_sum;
    int j;
    xx[idx] = curand_uniform(&localState)*5;
    xx_sum = reduce_sum(idx, xx[idx]);
    E_d[0] = xx_sum/N;
    for (int t = 1; t < T; t++) {
        xx_idx = xx[idx];
        g = eval_g(xx_idx, t);
        g_sum = reduce_sum(idx, g);
        p[idx] = g/g_sum;
```

```
        xx_prev[idx] = xx_idx;
        __syncthreads();

        j = discrete_rand(&localState, p);
        xx_idx = eval_f(&localState, xx_prev[j]);
        xx[idx] = xx_idx;
        xx_sum = reduce_sum(idx, xx_idx);

        // write the result to global memory from a single thread
        if (idx == 0) {
            E_d[t] = xx_sum/N;
        }
    }
}

__global__ void init_state(curandState *state, unsigned long seed)
{
    int idx = threadIdx.x;
    curand_init(seed, idx, 0, &state[idx]);
}

void print_E(float *E)
{
    for (int i = 0; i < T; i++) {
        printf("%f ", E[i]);
    }
    printf("\n");
}

void save_E(float *E)
{
    FILE *fp;
    fp = fopen ("E.dat", "w+");
    for (int i = 0; i < T; i++) {
        fprintf(fp, "%f\n", E[i]);
    }
    fclose(fp);
}

int main()
{
    curandState *state;
    cudaMalloc((void**)&state, N*sizeof(curandState));

    float E[T] = {0};
    float *E_d;
    const int size = T*sizeof(float);
    cudaMalloc((void**)&E_d, size);
    cudaMemcpy(E_d, E, size, cudaMemcpyHostToDevice);
```

```
    dim3 dimBlock(N);
    dim3 dimGrid(1);
    init_state<<<dimGrid, dimBlock>>>(state, 1);
    cudaDeviceSynchronize();

    asmc_bpf<<<dimGrid, dimBlock>>>(state, E_d);

    cudaMemcpy(E, E_d, size, cudaMemcpyDeviceToHost);
    cudaFree(E_d);
    cudaDeviceSynchronize();

    print_E(E);
    save_E(E);

    return EXIT_SUCCESS;
}
```