Simulation of Analogue and van Vleck Corrected Digital Correlators on GPU using OpenCL Framework

Leonid Benkevitch, Kazunori Akiyama
MIT Haystack Observatory, Westford MA
benkev@mit.edu

ABSTRACT

The purpose of writing this software is demonstration of the van Vleck correlation correction method over a wide range of the correlation values, from 0.001 or lower up to 0.999 or higher. In the simulation two independent random sequences of 64-bit double precision floating point numbers are generated. The sequences are sampled from the Gaussian distribution. A third random sequence is additively mixed in the both former in a proportion that guarantees a specified value of their correlation. The two correlated sequences are quantized with a software ADC. The "analog" sequences and their quantized counterparts are fed to the software correlator, and then the quantized correlation value is van Vleck corrected. Ideally, the analog and the van Vleck corrected correlations must be equal.

1. Why GPU and why OpenCL?

The GPU computation is necessary because the error in the correlation r is $(1 - r^2)/\sqrt{N}$, so for weakly correlated sequences keeping the error within 1% requires very large sequence lengths N. For example, correlating of one couple of sequences with the correlation 0.001 on a modern PC took over 11 hours. Conversely, using the NVIDIA videocard with GPU took only 3-4 minutes.

The simulations were carried out using the CUDA framework, which is aimed at the NVIDIA's hardware only. However, there are other GPU architectures on the marked. For example, the AMD GPUs sometimes provide more computational power and more memory at the same or lower price. The AMD GPUs are programmed in the OpenCL framework, which is universal: it provides the level of abstraction including virtually any GPU architecture. OpenCL allows writing portable and scalable codes for AMD, NVIDIA, ARM, various DSP hardware. This if why our effort to employ OpenCL for the simulation on both NVIDIA and AMD GPUs.

2. Convenience of OpenCL and PyOpenCL

3. Demo programs

The algorithms are obtained from the webpage by Prof. Sebastiano Vigna "Xoshiro / xoroshiro generators and the PRNG shootout" at `http://xoshiro.di.unimi.it/`.

Files with 'bits64' in their names: generators of 64-bit unsigned integers.
Files with 'unif64' in their names: generators of double precision.
floating-point numbers uniformly distributed over the [0..1] interval.
Files with 'norm64' in their names: generators of double precision floating-point numbers with normal (Gaussian) distribution having the mean at 0 and the standard deviation equal 1.

`rand_lib.c`: functions to initialize the states of PRNG.

Compilation of the demo programs:

```
$ gcc -std=c99 rand_lib.c gpu_rand_unif64.c
    -o gpu_rand_unif64 -l OpenCL
$ gcc -std=c99 cpu_rand_unif64.c
    -o cpu_rand_unif64
```

```
$ gcc -std=c99 rand_lib.c gpu_rand_bits64.c
    -o gpu_rand_bits64 -l OpenCL
$ gcc -std=c99 cpu_rand_bits64.c
    -o cpu_rand_bits64
```

`rand_lib.c`: C codes of random number generator xoshiro256** 1.0. The algorithms are obtained from the webpage by Prof. Sebastiano Vigna "Xoshiro / xoroshiro generators and the PRNG shootout" at `http://xoshiro.di.unimi.it/`

`pyrandl.pyx`: Module pyrandl contains class PyRandl. It provides random number generator xoshiro256** 1.0 written in Cython for speed.

Compilation in IPython:
`import pyximport; pyximport.install()`.
Or, use (with python 2)
`python setup.py build_ext --inplace`

`cpu_rand_bits64.c`, `cpu_rand_bits64_oneil.c`, `cpu_rand_bits64.py`: Generation on CPU of 64-bit long uniform integer numbers using the xoshiro256** 1.0. generator.

`cpu_rand_unif64.c`: Generation on CPU of 64-bit double uniform floating point numbers using the xoshiro256** 1.0. generator. Normalization to [0,1) via division by `ULONG_MAX`.

`gpu_rand_bits64.c`: Generation on GPU of 64-bit long int uniform floating point numbers. Uses PyRandl on CPU to generate initial states `rndst` for all the GPU threads. It uses `ker_rand_bits64.cl`: The GPU kernel.

`gpu_rand_unif64.c`, `gpu_rand_unif64.py`: Generation on GPU of 64-bit long int and double uniform floating point numbers. Use PyRandl on CPU to generate initial states `rndst` for all the GPU threads. Both use `ker_rand_unif64.cl`: The GPU kernel.

`simulate_vvcorr.c`: Incomplete. The OpenCL C codes happen to be too long and prone to errors compared with the PyOpenCL Python codes, so we switched to PyOpenCL.

`gpu_rand_norm64.py`: Generation on GPU of 64-bit double standard normal floating point numbers. Uses PyRandl on CPU to generate initial states `rndst` for all the GPU threads. Uses `ker_rand_norm64.cl`: The GPU kernel.

Obtaining double precision random numbers on GPU using double precision arithmetic operation is too slow, because double operations are an order of magnitude (or even more!) slower than the single, 32-bit operations. Also, the division as the way to normalize long int into double in [0,1) interval is slow, too. The idea is to generate two 32-bit adjacent ints and convert them into a single uniform double. A lot of useful information on the subject, as well as food for the brain I found in the webpage "Experilous: Perfect and Fast Random Floating Point Numbers."
`https://experilous.com/1/blog/post/perfect-fast-random-floating-point-numbers`.
The result is in the codes described below.

`cpu_rand_unif32.c`: Generation on CPU of 64-bit double uniform floating point numbers using the 32-bit xoshiro128** 1.0. generator. Normalization to [0,1) without the division. Initially, this program was used to generate 32-bit floats. Now the 32-bit float codes are commented out, but can be recovered.

4. Simulation programs

For convenience, the CPU part of the correlation and van Vleck correction simulation program has been written in Python with the use of the `pyopencl` module.

`ocl_calc_vvcorr.py`: The simulation program. It runs the GPU kernel `ker_calc_vvcorr.cl`.

The program only prints out the analogue and digital correlations. It has not been finished yet.

5. Current state and future work

The divisionless method based on 32-bit xoshiro128** has been tested on CPU in `cpu_rand_unif32.c`. I am trying to port it to GPU. I left intact the python and cl-kernel simulation programs, renaming them into `ocl_calc_vvcorr_xoshiro256starstar.py` and

`ker_calc_vvcorr_xoshiro256starstar.cl`.
They work well. The new couple, with 32-bit arithmetic,
`ocl_calc_vvcorr_xoshiro128starstar.py` and
`ker_calc_vvcorr_xoshiro128starstar.cl`,
is still being debugged. I should write the init script for 4x32-bit state. Maybe, in OpenCL to be run on GPU.

In future, instead of xoshiro128** 1.0 algorithm, I am going to use xoshiro128+ 1.0. Unlike xoshiro128**, which uses the multiplication operation twice per a number, xoshiro128+ uses no multiplications at all. Since the lowest four bits of its 32-bit output have low "linear complexity", they can be thrown away with the right shift. The double float number requires 52 bits for its mantissa, so $2 \times 28 = 56$ bits left from two 32-bit random numbers will be enough to form one uniform double.

The simulation program will be completed by analogy with its CUDA counterpart in
`/home/benkev/experimental/CUDA/cuda_sim_corr/`.
It lacks van Vleck function integration and its slope calculations in Python.