CMSC 691
High Performance Distributed Systems

GPUs for Evolutionary Computation
Random Number Generation

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

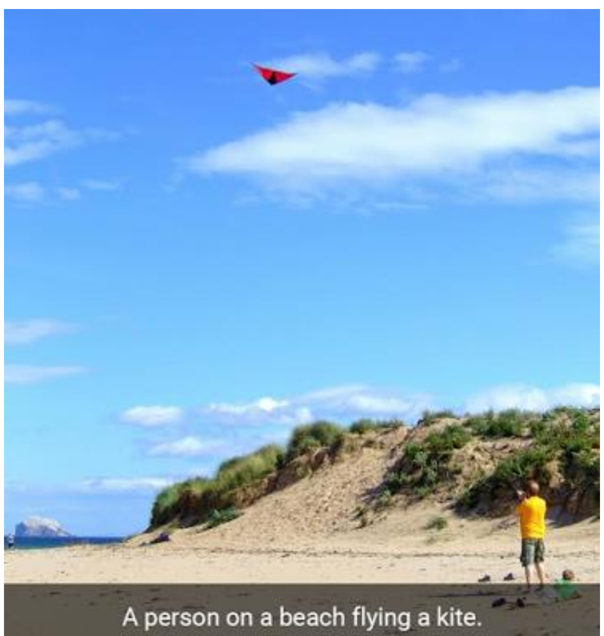
Massively parallel evolutionary computation on GPUs

- In Lecture #4 we introduced Evolutionary Algorithms and their parallelization using threads and RMI
- CPU-based parallelization is limited even when distributing workload in a cluster of compute nodes
- GPU computing provides a massively parallel solution!!!
- Advantages:
  - Afford many more resources for better accuracy
  - Reduce runtime for a given problem
  - Address much bigger dimensionalities
  - Deep learning, Computer Vision, NLP frameworks publicly available

Open source frameworks!

- Google Brain team & TensorFlow released last 09/22/2016

  Show and Tell: A Neural Image Caption Generator
- 2 weeks training phase using NVIDIA Tesla K20m GPU, datasets with millions of images, 93.9% accuracy



Random number generation in the GPU

- Stochastic methods such as evolutionary algorithms require high quality pseudorandom number generator (PRNG) or quasirandom number generator (QRNG)
- GPU threads cannot call host functions then *rand()* is not available
- cuRAND for fast random generation in host and device API
- Host API: a number of random numbers are generated and stored for alter use in a kernel
- Device API: random numbers are generated and immediately used in real time on an as-need basis
- Compile linking the library: -lcurand

cuRAND host API

1. Create a new generator with *curandCreateGenerator()*
2. Set the generator options, e.g. set the seed using *curandSetPseudoRandomGeneratorSeed()*
3. Allocate memory for the random numbers with *cudaMalloc()*
4. Generate random numbers with on or more calls to *curandGenerate()* or another generation function
5. Clean up the generator with *curandDestroyGenerator()*
6. Clean up everything else with *free()* and *cudaFree()*

Random number generator types

- Pseudorandom number generators
  - CURAND_RNG_PSEUDO_XORWOW
  - CURAND_RNG_PSEUDO_MRG32K3A
  - CURAND_RNG_PSEUDO_MTGP32
  - CURAND_RNG_PSEUDO_PHILOX4_32_10
  - CURAND_RNG_PSEUDO_MT19937
- Quasi-random number generators
  - CURAND_RNG_QUASI_SOBOL32
  - CURAND_RNG_QUASI_SOBOL64
  - CURAND_RNG_QUASI_SCRAMBLED_SOBOL32
  - CURAND_RNG_QUASI_SCRAMBLED_SOBOL64

Generator functions

- ***curandGenerate**(curandGenerator_t generator, unsigned int *outputPtr, size_t num)*
- ***curandGenerateUniform**(curandGenerator_t generator, float *outputPtr, size_t num)*
- ***curandGenerateNormal**(curandGenerator_t generator, float *outputPtr, size_t n, float mean, float stddev)*
- ***curandGenerateLogNormal**(curandGenerator_t generator, float *outputPtr, size_t n, float mean, float stddev)*
- ***curandGeneratePoisson**(curandGenerator_t generator, unsigned int *outputPtr, size_t n, double lambda)*

cuRAND device API

- *curandState_t* to keep track of the state of the random sequence
- Kernel threads call *curand_init(curandState_t *state)* to initialize the state of the random number generator
- Call *curand()* or one of its wrapper functions to generate pseudorandom or quasi random numbers as needed

Device distributions:

- *curand_uniform (curandState_t *state)*
- *curand_normal (curandState_t *state)*
- *curand_log_normal (curandState_t *state, float mean, float stddev)*
- *curand_poisson (curandState_t *state, double lambda)*

Evolutionary algorithms

- Individual representation using arrays
- Initialization of the population
  - Random initialization of each gene for each individual
- Genetic operators
  - Crossover and mutation, recombination of the genotypes
- #threads = #individuals x #dimensionality
- Fully coalesced memory access pattern
- Maximum GPU occupancy
- Next we'll see an application study to LSGO using distributed GPUs

CMSC 691
High Performance Distributed Systems

GPUs for Evolutionary Computation
Random Number Generation

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu