Stretch Move OCL Readme

Alex Kaiser
adkaiser@gmail.com

April 27, 2013

1 Introduction

This document describes Stretch Move OCL, a software package for Markov chain Monte Carlo sampling. The package is suitable for sampling generic continuous probability distributions. The algorithm for MCMC is called the Stretch Move, and is described in detail in section 3. It can be used for parameter estimation, integration, or any generic problem where sampling is required.

The code includes high performance parallel implementations of this algorithm. The package is written in C with OpenCL, which allows for control of the GPU (graphics processing unit) or other devices. It is designed to offer good parallel performance on difficult sampling problems.

Care has been taken to make the package easy to use. The user provides generic C code to evaluate a pdf, adjusts a few parameters and the code does the rest. To use the package effectively, the user needs nearly zero knowledge of parallel programming, OpenCL or GPU hardware .

2 Quickstart

Follow these steps to get started quickly with the included examples.

1. Place generic C code to evaluate the log of the probability density function (pdf) you would like to sample in "pdf.h."
2. Pick one of the included examples in "stretch_move_main.c" to run. Check that parameters in the example match your problem, which usually means setting the correct dimension.
3. Build the code with the included makefile.
4. Run the executable. The code will prompt to select an OpenCL implementation and hardware, then run the sampler.

3 The Stretch-Move MCMC algorithm

The MCMC algorithm included in this package is called the Stretch Move. It was developed at the Courant Institute of Mathematical Sciences by Jonathan Goodman and Jonathan Weare in 2010, see [1] for more details. The algorithm has a similar structure to Metropolis, and still uses a proposal and an accept/reject step. In this algorithm, we use a group or *ensemble* of sequences, also called *walkers*, since each one is proceeding in its own random walk. Each walker on each iteration

of the algorithm represents a valid sample of the distribution. Thus, on each iteration the algorithm generates multiple samples. To update one walker, another walker is randomly selected from the *complementary ensemble*. A move is then proposed alone a line between the current walker k and the complementary walker j according to the distribution

$$Y = X(j) + z(X(k) - X(j))$$

and then accepted or rejected. This generally gives better autocorrelation time as compared to other MCMC methods.

The algorithm is efficient. It requires a comparable amount of arithmetic to update a single walker to other MCMC methods. Crucially, the algorithm can be parallelized. To accomplish this, split the walkers into two groups X_{red}, X_{black}. If there are K total walkers, then the algorithm allows precisely $K/2$ updates to be performed in parallel. This splitting allows the detailed balance condition to remain satisfied when the walkers are updated in parallel.

The algorithm is *affine invariant*, which means that for A, a linear transformation, and vector b, sampling a PDF $g(x) = Af(x) + b$ is equivalent to sampling f then applying A and b after. Heuristically, this means that the algorithm is not sensitive to skewed distributions.

The algorithm requires an auxiliary random variable Z with pdf $g(z)$, which must satisfy $g(1/z) = (1/z)g(z)$. This distribution is usually $g(z) = 1/\sqrt{2z}$ on $(1/2, 2)$, and this is the only distribution used in this project for now.

A description of a single iteration of the algorithm to updated one walker is as follows. All of the walkers in each group can be updated completely in parallel.

To update $X_{red}(k)$

- Randomly select a walker from the complementary ensemble $X_{black}(j)$
- Sample $z \sim g(Z)$.
- Compute a proposed move Y

$$Y = X_{black}(j) + z(X_{red}(k,t) - X_{black}(j))$$

- Compute a likelihood

$$q = z^{N-1}\frac{f(Y)}{f(X_{red}(k,t))}$$

- If $q > 1$, accept. Otherwise, accept with probability q.
- If accept $X_{red}(k, t+1) = Y$, else $X_{red}(k, t+1) = X_{red}(k, t)$.

The additional factor of z^{N-1} in the likelihood ratio is to ensure that detailed balance is satisfied.

4 A Short Description of GPUs and OpenCL

This package is written in C and OpenCL. OpenCL is a set of extensions to C that allow the user to access the GPU (graphics processing unit) for general computations. This section contains a very short summary of vocabulary and basic issues encountered in OpenCL, and ends with a description of what a user needs to know to use the package.

The program execution in controlled by a standard CPU program. The CPU is also referred to as the *host*, and the GPU as the *device*. An OpenCL program that runs on the device is called a *kernel*. The device may not be a GPU, as OpenCL also runs on other types of hardware, but we will focus on the GPU here.

A GPU has a lot of parallelism on a small, relatively inexpensive chip; it can execute many instructions simultaneously. It has a large piece of RAM called global memory, and smaller, faster pieces of memory called local memory and registers. It has an interconnect to CPU memory which, on modern chips, is high-bandwidth, but also high latency.

A GPU thread is called a *work-item*. OpenCL automatically manages the parallel execution of work-items on the GPU. The work-items run in a group of threads called a *work-group*. The execution of all work-items in a group is SIMD, or "single instruction, multiple data." This means the work-items in a work-group execute the same instructions simultaneously, in parallel. Instructions that would not be executed in a serial program (for example, an else clause in an if statement that evaluates to true) are still executed, but their values are thrown away. This means that branches and loops which run for a variable number of iterations may make performance slow. The GPU can also perform latency hiding, which means that it can execute instructions while waiting for memory reads and writes to complete. There is lots more about this in the literature and online.

An included package called Ranluxcl, see [2], is included for generating random numbers.

Here's what this means for using the package. Use a big enough work-group to get SIMD parallelism, that means hundreds. Use enough walkers to keep the GPU busy, that means thousands or tens of thousands of walkers. Anything that runs for a variable amount of time will run as slow as the slowest if your work group. It may be worth working to make sure a Newton's solve, for example, finishes in a small number of iterations. If performance is still bad, consider more sophisticated memory management or other optimizations. See section 6 for more advice.

5 Using the code

5.1 Building

The code uses generic C with OpenCL extensions and library calls. You must have OpenCL and a C compiler installed to use this package, but there are no other external requirements. A generic makefile is supplied that should work on most Unix or Linux machines, including OSX.

5.2 Examples

The file "stretch_move_main.c" includes two examples of using the sampler. The first shows the simplest possible example of running the sampler. The second uses more features of the code. It

uses data, run burn-in and performs sampling. Is computes basic statistics including autocorrelation time, calculates histograms and outputs that data to a file.

5.3 Setting things up

First, you need to specify the pdf you want to sample. For numerical reasons, the code requires the logarithm of the pdf, rather than the pdf itself. The pdf does not need to be normalized. To do this, place your code in the file "pdf.h." The pdf function is written standard C, with a few restrictions. There is no access to I/O, so all data must be passed to the function in memory rather than read from a file. There's also no dynamic memory allocation, recursion or function pointers.

Suppose one wanted to sample an N dimensional Gaussian with pdf

$$f_X(x) \propto \exp\left(-\frac{1}{2}\sum_{i=0}^{N}(x_{i+1}-x_i)^2\right).$$

where by definition $x_0, x_{N+1} = 0$. Then the log pdf function would be implemented as

```
float log_pdf(float *x, data_struct data_st, const __local float *data){
    float sum = x[0]*x[0] + x[NN-1]*x[NN-1];
    for(int i=0; i<NN-1; i++){
        sum += (x[i+1] - x[i]) * (x[i+1] - x[i]);
    }
    return -0.5f * sum;
}
```

The dimension "NN" is set up at kernel compile time by the initialization routine and is always available in this function. The data structure "data_st" and array "data" are passed even if they are not used. This is for generality, since the package handles all the data movement.

Next, define needed constants in "constants.h." These may define the dimension of the problem or observations, values for the prior, loop bounds, or anything else.

Also, if the PDF requires additional data, add the types to "data_struct.h." Because of OpenCL rules, you must also specify this structure in "pdf.h" for the kernel to use. By default this contains a float, you may add any scalars or small statically defined arrays. Note that OpenCL requires the scope of such a struct to be private.

If you wish to include large arrays, use the buffer "data." This is a single float array, and all the arrays must be packed into this array. OpenCL is not flexible to complicated structures involving variable numbers of pointers or pointers to pointers. The best general structure is to insist that the user gets one data array, and should unpack it manually.

Consider sampling a multivariate normal with mean $\vec{\mu}$ and covariance matrix Σ. The pdf is given

$$f(\vec{x}) \propto \exp\left[-\frac{1}{2}(\vec{x}-\vec{\mu})\Sigma^{-1}(\vec{x}-\vec{\mu})\right].$$

We want to pass a vector "mu" of length NN for the mean, and an NN by NN matrix for the inverse covariance "inv_cov." Set these to be contiguous in memory in the array "data" which is passed

to the initialization routine. For example, allocate "data" to be a length NN + NN^2 float array. Then set

```
data[0] = 0.5f;                          // first component of mean vector
...
data[NN-1] = 10.0f;                      // last component of mean vector
data[0 + 0*NN + NN] = 2.0f;              // first entry of matrix
                                         // matrix starts at entry NN
...
data[NN-1 + (NN-1)*NN + NN] = 3.0f;      // last entry of matrix
```

The package handles moving the packed array to the device. In your code for the pdf, set pointers or otherwise read the data. For example:

```
// The first NN elements of the data array are the means
const __local float *mu = data;

// NN elements later is the inverse covariance matrix
const __local float *inv_cov = data + NN;
```

Now these pointers can be used as normal arrays to evaluate the pdf. If you unpack data this way, it must have the same scope and address space as the data array. Use pointers rather than reallocate and copy to avoid wasting time on memory operations. This is illustrated in full in the second included example.

5.4 Sampling

Now that the pdf and definitions are are in place, initialize a sampler object. Set up some parameters for your run, start with the chain length. This is the number of ensemble samples that will be run when you start the sampler.

```
cl_int chain_length = 100000;
```

Next is the dimension of the problem:

```
cl_int dimension = 10;
```

Set the size of each half of the ensemble, which also corresponds to the total parallelism available:

```
cl_int walkers_per_group = 1024;
```

Last, set the work group size, which must divide the number of walkers per group.

```
size_t work_group_size = 128;
```

Call the initialization routine to allocate all the necessary arrays and compile the OpenCL kernels.

```
sampler *samp = initialize_sampler(chain_length, dimension,
                      walkers_per_group, work_group_size,
                      0, 0, NULL, CHOOSE_INTERACTIVELY, CHOOSE_INTERACTIVELY);
```

Use these default values for the last few parameters.

Then run the burn-in for 10000 steps:

```
run_burn_in(samp, 10000);
```

The sampler is now ready. Run it:

```
run_sampler(samp);
```

The array `samp->samples_host` is now filled with samples ready for use. Samples are stored in "component major" order, so to access component `i` of sample `j`, use

```
samp->samples_host[i + j*samp->N];
```

That's it for basic usage. The included examples contain also illustrate more features of the package, including generating histograms and basic statistics.

6 Questions and issues

Here are some issues that you may run across in using the code.

- What are walkers and how should I set them up?

  The algorithms run with an ensemble of random walks, which itself is a Markov chain. On every iteration after convergence, each walker is an independent sample of the invariant distribution. For the algorithm to be valid, you must have more walkers than the dimension of the problem. The walkers must span the space, in a linear algebra sense, so they cannot lie in a hyperplane that is a proper subset of the space. Generally, for performance you want to use a lot of walkers, think thousands or tens of thousands.

  Default initialization is to put a random uniform(0,1) value in each component of each walker. If your walkers are all very far from a region with lots of probability mass, they may move completely randomly or not move at all. If there is a region with zero probability, and the walkers are initialized there, the algorithm will almost certainly have trouble converging. If you think you know where the interesting regions of probability mass are, you can set the walkers to start near there. This is a risk, because they may not leave and explore the rest of the space.

- My sampler is not converging and my histograms are noise.

  Some suggestions:

  - Test and debug your pdf. Subtle errors in the pdf may produce very subtle wrong answers, or may overwhelm the sampler and just give noise.
  - Try running simulated annealing as shown in the second example. Run for a lot of steps.
  - Run the burn-in for very long.
  - What is the dimension of your problem? Experiments have shown that in high dimensions or with very many walkers a very long burn in may be required. If the dimension is extremely large, the size of the state space may be so large that convergence may not occur at all.

- What is this data packing thing?

  As mentioned above, OpenCL does not handle structures involving variable numbers of pointers or pointers to pointers. This is the best compromise that allows flexibility for users. Note that there are better solutions, but they require modifying much more of the code. See the following question and email for help.

- I have a complicated data structure that I need to evaluate my likelihood function. I hate packing my data. What should I do?

  This must be handled manually. Modify the current code and add arguments. Email for help. The second is to pack everything into a single float array. In your pdf, manually unpack your data. If you have packed many arrays together,

- How do I check what GPUs are available on my system?

  If you run the initialize kernel with the "CHOOSE_INTERACTIVELY" option the code will output lists of the available platform. Select one from the list, and the compatible devices will be listed as well.

- My performance is terrible.

  Some suggestions:

  - How many walkers are you using? Try using more. Think 2^{10}, 2^{14} or more as memory allows. Make sure you run lots of burn-in when using so many walkers.
  - Are you running a GPU? The code may give decent performance running threaded on CPUs, but it may not.
  - Now that you're on the GPU, what is your work group size? It should decently big, try 128 or 256 or even larger. Powers of two are good.
  - Is the GPU also running your monitor at the same time? This will make everything slow. Laptop GPUs are frequently slow.
  - Is the time to evaluate your pdf highly variable? If you have, for example, an iterative solve like Newton's method, the SIMD nature of the GPU means all your work-items (GPU threads) run as slow as your slowest work-item. If you can improve the worst case performance you may improve the overall very much.
  - Is the time to evaluate you pdf very fast? You may be spending all your time waiting for memory to be sent to the host. This is not very common.

- Why are these cl_whatever datatypes all over the place?

  These datatypes guarantee a particular size and alignment to the compiler. Use them or risk difficult to diagnose, very confusing bugs. They cover the full set of basic C types and more, so you might as well.

- I get an error saying "invalid arg size." What is this?

  In OpenCL, data types have to be the specified width and have correct alignment. This means that you must match the data exactly — if the kernel calls for an int, you should pass a "cl_int" and not a "cl_long." Passing an int value to a function which expects a long int frequently doesn't matter in C. This is not true in OpenCL and will cause bugs. This is a good reason to use OpenCL datatypes throughout your C code.

- I get an error regarding local memory.

  The current code places the data array in local memory for speed. If your data is too big, then it needs to be kept in global. This needs a manual change to the code. In the (near) future, this can be changed by a definition switch once I debug the feature.

References

[1] Goodman, J., and Weare, J. Ensemble samplers with affine invariance. *Commun. Appl. Math. Comput. Sci. 5* (2010), 65–80.

[2] Nikolaisen, I. U. ranluxcl v1.3.1, 2011. https://bitbucket.org/ivarun/ranluxcl/.