Leveraging GPU Libraries for Efficient Computation of Bayesian Spatial Assignment Models in R

Colin Rundel

University of California, Los Angeles / Duke University

August 1, 2012

Project Background

Developing methods to make use of intrinsic markers (genetic and isotopic signals) for the purpose of inferring migratory connectivity.

- Existing methods are too coarse for most applications
- Large amounts of data are available ($>$150,000 feather samples from $>$500 species)
- Genetic assignment methods are based on Wasser, et al. (2004)
- Isotopic assignment methods are based on Wunder, et al. (2005)

Preliminary Data (microsats and $\delta^2\mathrm{H}$):

- Hermit Thrush (*Catharus guttatus*) - 138 Individuals, 14 Locations, 6 Loci, 9-27 Alleles
- Wilson's Warbler (*Wilsonia pusilla*) - 163 Individuals, 8 Locations, 9 Loci, 15-31 Alleles

Allele Frequency Model

For the allele i, from locus l, at location k

$$\boldsymbol{y_{l\cdot k}} \sim \mathsf{MN}\left(n_{lk} = \textstyle\sum_i y_{lik}, \boldsymbol{f_{l\cdot k}}\right)$$

$$f_{lik} = \frac{\exp(X_{lik})}{\sum_i \exp(X_{lik})} \qquad \boldsymbol{X}_{li} \sim \mathcal{N}(\boldsymbol{M}_{li}, \boldsymbol{\Sigma})$$

Allele Frequency Model

For the allele i, from locus l, at location k

$$\boldsymbol{y_{l\cdot k}} \sim \mathsf{MN}\left(n_{lk} = \textstyle\sum_i y_{lik}, \boldsymbol{f_{l\cdot k}}\right)$$

$$f_{lik} = \frac{\exp(X_{lik})}{\sum_i \exp(X_{lik})} \qquad \boldsymbol{X}_{li} \sim \mathcal{N}(\boldsymbol{M}_{li}, \boldsymbol{\Sigma})$$

Likelihood:

$$\begin{aligned} &\prod_l \prod_k \frac{n_{lk}!}{\prod_i y_{lik}!} \textstyle\prod_i (f_{lik})^{y_{lik}} \\ \times &\prod_l \prod_i 2\pi^{-r/2} |\boldsymbol{\Sigma}|^{-1/2} \exp\left[-\frac{1}{2}(\boldsymbol{X_{li}} - \boldsymbol{M}_{li})' \boldsymbol{\Sigma}^{-1} (\boldsymbol{X_{li}} - \boldsymbol{M}_{li})\right] \\ \times &\ \pi(\boldsymbol{\theta}) \end{aligned}$$

Implementation

Model fitting and prediction is done via MCMC

- Original implementation in pure C++ with minimal dependencies
- Rewritten using R / C++ via Rcpp(Armadillo)
  - Code closer to matrix notation (and R)
  - Transparent use of high performance LAPACK implementations (ATLAS, OpenBLAS, Intel MKL)
- GPU based optimizations were added using CUDA, CUBLAS, and MAGMA libraries
- Cross platform R package scatR (hopefully added to CRAN soon)

Performance

System specs - 4 core Intel i5-2500K, GeForce GTX 460

Software specs - Ubuntu 12.04, ATLAS 3.8.4, Cuda 4.2, Magma 1.1

Performance

System specs - 4 core Intel i5-2500K, GeForce GTX 460
Software specs - Ubuntu 12.04, ATLAS 3.8.4, Cuda 4.2, Magma 1.1

Performance during model fitting is quite good...

100,000 iterations in ~ 45 seconds

Performance

System specs - 4 core Intel i5-2500K, GeForce GTX 460
Software specs - Ubuntu 12.04, ATLAS 3.8.4, Cuda 4.2, Magma 1.1

Performance during model fitting is quite good...

100,000 iterations in ~ 45 seconds

Performance during prediction is much slower...

100,000 iterations in ~ 2580 seconds (43 mins)
(predictions calculated every 100 iterations)

Performance

System specs - 4 core Intel i5-2500K, GeForce GTX 460
Software specs - Ubuntu 12.04, ATLAS 3.8.4, Cuda 4.2, Magma 1.1

Performance during model fitting is quite good...

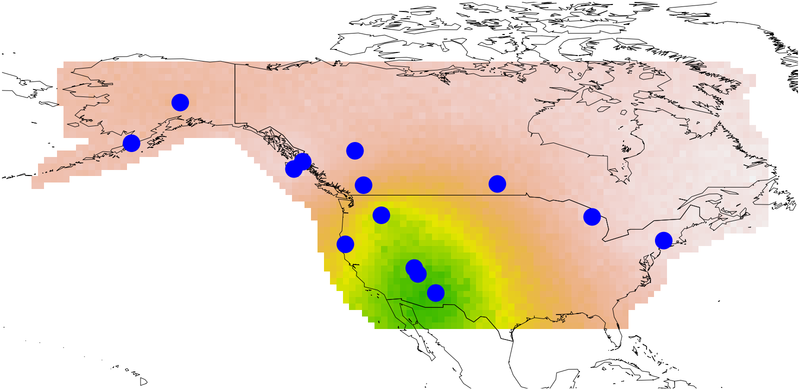
100,000 iterations in ~ 45 seconds

Performance during prediction is much slower...

100,000 iterations in ~ 2580 seconds (43 mins)
(predictions calculated every 100 iterations)

Not too bad in the greater scheme of things, but we would really like to be able to do cross validation (~ 200 runs per species) ...

Prediction Example



Prediction algorithm details

Why is the prediction slow?

Prediction algorithm details

Why is the prediction slow? We are predicting allele frequencies for Hermit thrush at 3318 novel locations.

To do so we need to draw samples from:

$$\boldsymbol{X_p}|\boldsymbol{X_m} \sim \mathcal{N}(\boldsymbol{\mu_p} + \boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}(\boldsymbol{X_m} - \boldsymbol{\mu}_m),\ \boldsymbol{\Sigma}_p - \boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}\boldsymbol{\Sigma}_{mp})$$

Prediction algorithm details

Why is the prediction slow? We are predicting allele frequencies for Hermit thrush at 3318 novel locations.

To do so we need to draw samples from:

$$\boldsymbol{X_p} | \boldsymbol{X_m} \sim \mathcal{N}(\boldsymbol{\mu_p} + \boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}(\boldsymbol{X_m} - \boldsymbol{\mu}_m),\ \boldsymbol{\Sigma}_p - \boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}\boldsymbol{\Sigma}_{mp})$$

Algorithm steps

1. Calculate $\boldsymbol{\Sigma}_{pm}$ and $\boldsymbol{\Sigma}_p$
2. Calculate $\boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}$
3. Calculate Chol$(\boldsymbol{\Sigma}_p - \boldsymbol{\Sigma}_{pm}\boldsymbol{\Sigma}_m^{-1}\boldsymbol{\Sigma}_{mp})$
4. Sample from MVN
5. Calculate allele frequencies
6. Output results

Prediction algorithm timings

	Step	CPU Timing (secs)
1.	Covariances	1.02
2.	$\boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}$	0
3.	Cholesky	1.15
4.	Sample from MVN	0.23
5.	Allele Freq	0.14
6.	Output	0
	Total	2.54

Prediction algorithm timings

	Step	CPU Timing (secs)	CPU+GPU (secs)
1.	Covariances	1.02	0.05
2.	$\boldsymbol{\Sigma}_{21}\boldsymbol{\Sigma}_{11}^{-1}$	0	0
3.	Cholesky	1.15	0.23
4.	GP Sample	0.23	0.06
5.	Allele Freq	0.14	0.14
6.	Write	0	0
	Total	2.54	0.48

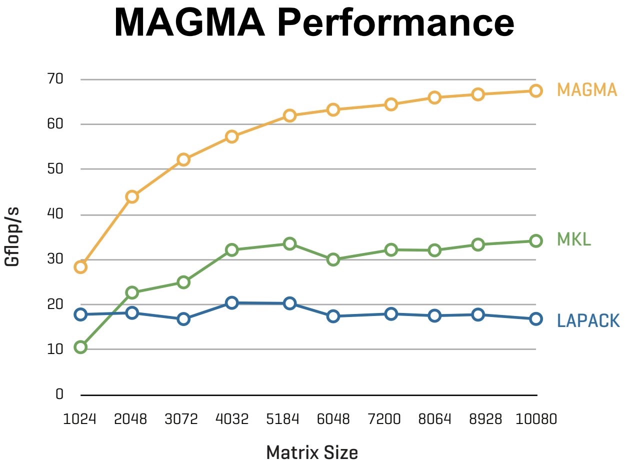
Improving the Cholesky step

Not surprising given Cholesky factorization is $\mathcal{O}(n^3)$ and $n = 3318$.

There isn't a magical solution to this, so we just want to use the fastest possible implementation of the Cholesky decomposition.

- **Intel MKL / OpenBLAS / Eigen** - all (multicore) CPU based with very marginal improvement
- **CUBLAS** - part of NVidia's CUDA toolkit, implements core BLAS functions (but not cholesky)
- **CULA** - proprietary / closed source (dense and sparse) GPU linear algebra library with an expensive license
- **MAGMA** - open source Multicore+GPU dense linear algebra library (CUDA and OpenCL implementations)

Cholesky GPU



Ltaief, H. "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators"

VECTAR'10 Presentation, Berkeley, CA, June 22-25, 2010.

Additional Considerations

- There are costs for moving data on to and off of the GPU
- Once the data is there, may as well do as many calculations as possible
  - Drawing sample from the GP is sped up by performing the matrix multiplication on the GPU
- GPU code is much more verbose / dense

Additional Considerations

- There are costs for moving data on to and off of the GPU
- Once the data is there, may as well do as many calculations as possible
  - Drawing sample from the GP is sped up by performing the matrix multiplication on the GPU
- GPU code is much more verbose / dense

Armadillo

```
arma::mat tmp = cov12.t() * p.Sinv
```

GPU (CUBLAS)

```
cublasDgemm_v2(
    p.handle, CUBLAS_OP_T, CUBLAS_OP_N,
    n_pred, n_known, n_known,
   &one,
    p.d_cov12, n_known,
    p.d_invcov11, n_known,
   &zero,
    p.d_tmp, n_pred
)
```

Improving Covariance calculations

Covariance in our model is assumed to be stationary and isotropic (depend only on distance between locations)

- Elements of the covariance matrix can be calculated independently
- Small scale “embarrassingly parallel” ⇒ good candidate for the GPU.
- Implementation is straight forward

Improving Covariance calculations

Covariance in our model is assumed to be stationary and isotropic (depend only on distance between locations)

- Elements of the covariance matrix can be calculated independently
- Small scale "embarrassingly parallel" ⇒ good candidate for the GPU.
- Implementation is straight forward

```
__global__ void powered_exponential_kernel(double* dist, double* cov,
                                           const int n, const int nm,
                                           const double sigma2, const double phi,
                                           const double kappa, const double nugget)
{
    int n_threads = gridDim.x * blockDim.x;
    int pos = blockDim.x * blockIdx.x + threadIdx.x;

    for (int i = pos; i < nm; i += n_threads)
        cov[i] = sigma2 * exp(-pow(dist[i] / phi, kappa)) + nugget*(i%n == i/n);
}
```

Summary

Relatively small changes in one function resulted in ~ 5x improvement

- Cross validation results in days and not weeks
- Started with trying to find an improved Cholesky decomposition, other optimizations followed
- GPU implementation was relatively painless
- Libraries are under active development (read: things can and will break)
- External libraries make package development non-trivial

Questions, Comments?

email	:	rundel@gmail.com
github	:	*http://github.com/rundel/*
presentation	:	*http://github.com/rundel/Presentations/*