1 Methods

1.1 Coordinate Descent

In case control studies of disease, affection status is related to a set of genetic and/or environmental covariates through a vector of regression coefficients, denoted as β. Suppose y stores the vector of binary values for affection status and x is an n by m matrix of covariates comprised of discrete genotypes (coded as the number of risk alleles) and/or environmental covariates (treated as continuous values), defined across n subjects and m SNPs. The probability of affection for subject i is defined as

$$p_i = \frac{1}{1 + \exp(-\beta^T x_i y_i)}. \tag{1}$$

Each element of β is interpreted as the log odds ratio of disease risk for each unit increase of its respective covariate. The log-likelihood across all observations is then

$$L(\beta) = \sum_{i=1}^{n} [y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \tag{2}$$

We adopt the penalized regression model described in [Zhou *et al.*, 2010] to incorporate both an L1 penalty and L2 penalized variable groups (e.g. genes, pathways). The L1 parameter (denoted λ_L) enforces sparsity, while the L2 parameter (denoted λ_E) encourages variables mildly correlated to a strong predictor within a group to enter the model. Our goal is to minimize the objective function

$$f(\beta) = -L(\beta) + \lambda_L ||\beta||_1 + \lambda_E \sum_G ||\beta_G||_2. \tag{3}$$

This parameterization is a generalization of several well-known methods. For example, setting $\lambda_E = 0$ reduces Eq 3 to the original LASSO [Tibshirani, 1996] whereas including all variables into the set G reduces Eq 3 to the elastic net [Zou and Hastie, 2005].

Regularized regression is usually solved using cyclic coordinate descent (CCD) [Zhang and J. Oles, 2001]. At each variable j, CCD updates β_j as

$$\beta_j^{new} = z_j = \beta_j - \frac{f'(\beta_j)}{f''(\beta_j)} \tag{4}$$

, where the first derivative is

$$f'(\beta_j) = \left.\frac{df(z_j)}{dz}\right|_{z=\beta_j}$$
$$= \sum_{i=1}^{n} x_{ij}y_i \frac{1}{1+\exp(\beta^T x_{ij}y_i)} - \lambda_L \mathrm{sgn}(\beta_j) - \lambda_E \begin{cases} \frac{\beta_j}{||\beta_G||_2}, & ||\beta_G||_2 > 0 \\ sgn(\beta_j) & ||\beta_G||_2 = 0 \end{cases} \quad (5)$$

and the second derivative is

$$f''(\beta_j) = \left.\frac{d^2 f(z_j)}{d^2 z}\right|_{z=\beta_j}$$
$$= \sum_{i=1}^{n} x_{ij}^2 \frac{\exp(\beta^T x_i y_i)}{1+\exp(\beta^T x_i y_i)^2} + \lambda_E \begin{cases} \frac{1}{||\beta_G||_2}(1 - \frac{\beta_j^2}{||\beta_G||_2^2}), & ||\beta_G||_2 > 0 \\ 0 & ||\beta_G||_2 = 0 \end{cases} \quad (6)$$

Each element update of the vector β updates the length n vector of fitted values $x\beta$. CCD traverses the set of variables over multiple cycles until a convergence criterion is reached (e.g. the likelihood improvement between cycles is sufficiently small). In a review article, Wu and Lange described a slight variation on CCD called greedy coordinate descent (GCD) [Wu and Lange, 2008]. Rather than update the vector β, GCD keeps track of the updated likelihood at index j after evaluating Eq 4. After evaluating all p variables, β is updated at the index corresponding to the largest increase in the likelihood. In contrast to CCD, which exposes parallelism across subjects, GCD offers the opportunity to carry out calculations in parallel across both subjects and variables.

1.2 OpenCL kernels

One feature that distinguishes our software from the majority of available GPU software releases, which is implemented in nVidia's proprietary CUDA library [nVidia, 2011], is our decision to develop parallel code using OpenCL [Khronos, 2011]. OpenCL's support across the key hardware manufacturers ensures that OpenCL programs will require minimal configuration changes when targeting a wide range of hardware, including GPU devices from ATI or nVidia, or massively parallel CPUs that are currently being developed by Intel and AMD.

OpenCL programs follow a heterogeneous computing model. That is, components of an algorithm that can execute in parallel on an SIMD device such as a GPU are implemented in routines, called kernels, that can be invoked by serial host code. Kernels execute data-parallel instructions so that at each clock cycle, calculations are processed in a lock-step manner across distinct *work-items*, an abstraction that maps computing cores to distinct memory addresses (e.g. registers, elements of an array). OpenCL defines an explicit memory model so data must

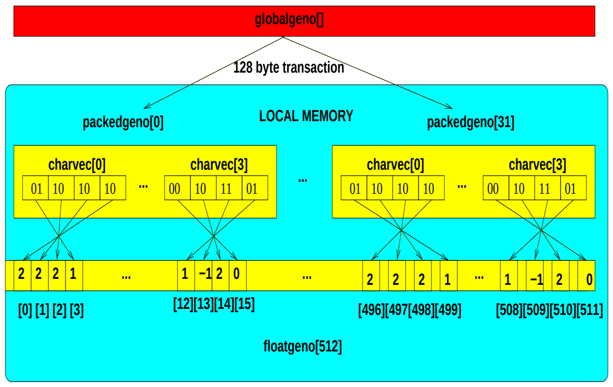

Figure 1: Decompressing genotypes from a single memory load transaction

explicitly be moved by programmers between layers of a memory hierarchy. On GPU devices, *global memory* is scoped across all work-items, *local memory* is scoped only within the *work-group* that a work-item is assigned to, and each *register* is scoped only to a specific work-item. Global memory is the most abundant memory space, while local memory and register space is far scarcer (64kB on high end devices). Global memory has up to two orders of magnitude greater latency than the latter two memory spaces, so good programming practice includes moving data into fast memory as early as possible during kernel routines.

It is vital that large datasets be stored efficiently in global memory, and compacted if possible, since even the most advanced GPU devices at the time of writing contain less than 2.5 GB of memory. Fortunately, SNP genotype data can be easily compressed, given the 3 possible cases plus "no-calls". One strategy is to declare genotypes in memory as a large array of character primitives, where each byte stores four genotypes. However, this strategy utilizes only 25% of the available bandwidth between global and local memory, since work-items fetch/store a minimum of 4 bytes when accessing global memory. Our approach is to declare an array of custom containers so that each container stores a cluster of four genotypes:

```
typedef struct {
  char geno[4];
}__attribute__((aligned(4))) packedgeno_t;
```

After transferring elements from a global array to a local array, data can quickly be decompressed using up to three bit shift operations and a single mask operation. The decompression step executes in parallel, storing results in a second local array of floats as illustrated in Figure 1. Hence, each set of 32 packed genotype elements, stored in global memory, decompress into 512 floating point genotypes in local memory.

As alluded to above, work-items are organized into work-groups, arranged in a three dimensional lattice called a work-group. Work-groups themselves are also organized in a three dimensional lattice. The dimensions of the

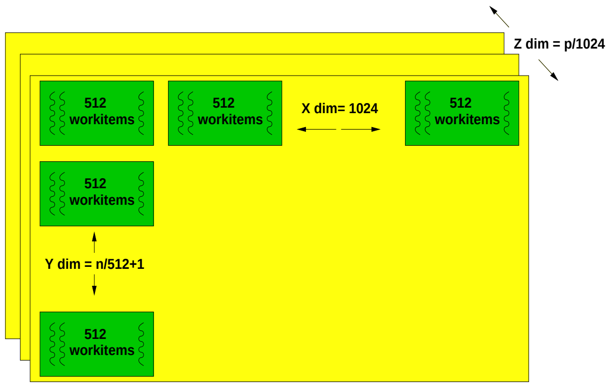

Figure 2: Work-group layout for GCD

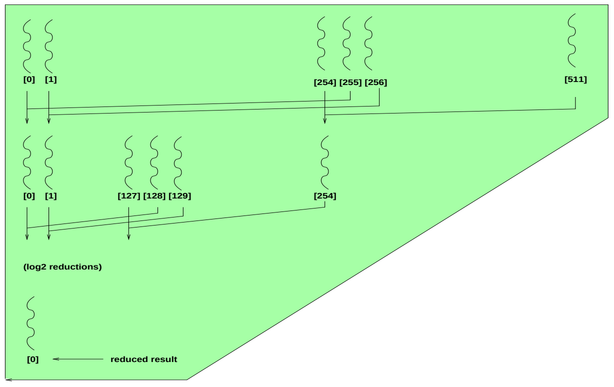

Figure 3: $\log_2$ reduction for summing array elements

lattices at both levels are defined by the programmer in order to make the most effective use of available cores and memory resources. On GPUs, instructions in work-groups are executed asynchronously and in any order. This leads to huge gains in efficiency as the level of parallelization increases, since memory latency in some work-groups is masked by calculations being made in other work-groups.

For parallel GCD, covariate values at a variable are divided into blocks of 512 elements, and stored in local memory so that each work-group can operate on a particular block of memory. Work groups compute contributions to statistics such as the likelihood (Eq 2), first (Eq 5), and second (Eq 6) derivative. Figure 2 illustrates how we organize work-groups and work-items for GCD. Work-items have an x-dimension of 512, and y,z dimensions of 1. Work-groups have a x-dimension of 1024 (representing blocks of the p variables), a y-dimension of $n/512+1$, and a z-dimension of $p/1024$.

Calculations that aggregate across elements of an array, such as summations or max-operators, can take advantage of moderate parallelization. A common practice in parallel programming is to perform a $\log_2$ reduction.

For any array of size that is a power of 2, elements in the second half of the array can be "collapsed" into the first half. The procedure is recursively repeated on the first half until the terminating condition at which the final value is stored in the first element. Figure 3 illustrates this concept.

1.3 Distributed MPI algorithm

For especially large problems such as impending GWAS-Seq data, it may be necessary to pool computing and memory resources from multiple GPU devices. OpenCL exposes an API that allows host code to access multiple devices on a host. Implementing MPI (message passing interface) routines achieves the same purpose but would seamlessly scale across multiple hosts. MPI is an open standard that enables multiple processors (locally or across hosts) to communicate amongst each other. Our MPI implementation is conceptually straightforward. The MPI master node is the entry point of the algorithm. After loading configuration and study data into memory, it partitions the design matrix of covariate values into submatrices, which are then transferred to all available slave nodes for initialization. Each slave node carries out parallel GCD on its own subproblem, followed by a max reduction step at the MPI master. The following pseudocode describes our algorithm more explicitly:

```
while not converged do
  Broadcast $j^*$: index of best variable from previous iteration
  for host = 1 $\rightarrow$ slavehosts, in parallel do
    $x\beta = x\beta + x_{-j^*}\Delta\beta_{j^*}$
    for all $j \in varblock[host]$, in parallel do
      for all $k \in subjectblocks$, in parallel do
        for all $i \in subjectblocks[k]$, in parallel do
          compute subject $i$'s contribution to sub-gradient,sub-hessian at variable $j$ using Eq 5 and Eq 6
        end for
      end for
      Sum over subject specific gradient and hessian via $\log_2$ reduction
    end for
    for all $j \in varblock[host]$, in parallel do
      for all $k \in subjectblocks$, in parallel do
        for all $i \in subjectblocks[k]$, in parallel do
```

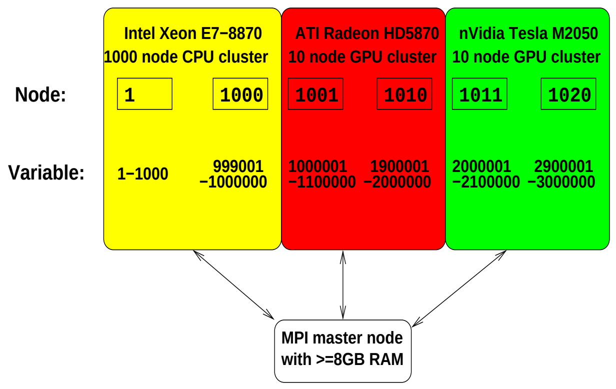

Figure 4: Using MPI to coordinate the LASSO optimization

```
                compute subject $i$'s contribution to likelihood at variable $j$ using Eq 2
            end for
        end for
        Sum over subject specific likelihood via $\log_2$ reduction
    end for
    Apply a max operator for variable with highest likelihood increase, store as best variable $j^*$
  end for
  Update $\beta_j^*$ on host
end while
```

Our MPI framework enables data and computations to be load balanced across multiple cores on a single desktop or processors on a large cluster, such as the heterogeneous cluster depicted in Figure 4. In this hypothetical example, the last 20 nodes, each assigned to fit 100,000 variables are OpenCL enabled, making use of massively parallel GPU resources; each of the first 1000 nodes fit far less variables, but could be moderately accelerated using parallel constructs optimized for multi-core CPUs such as OpenMP [OpenMP, 2011].

1.4 Rare variant simulation

The Genetic Analysis Workshop 17 [Ziegler *et al.*, 2011] provided participants with genotypes across 24,487 SNPs on 697 individuals, taken from Pilot 3 of the 1000 Genomes Project. The genotypes were called using the Unified Genotyper method of the Genome Analysis Toolkit (GATK) package [DePristo *et al.*, 2011]. For our

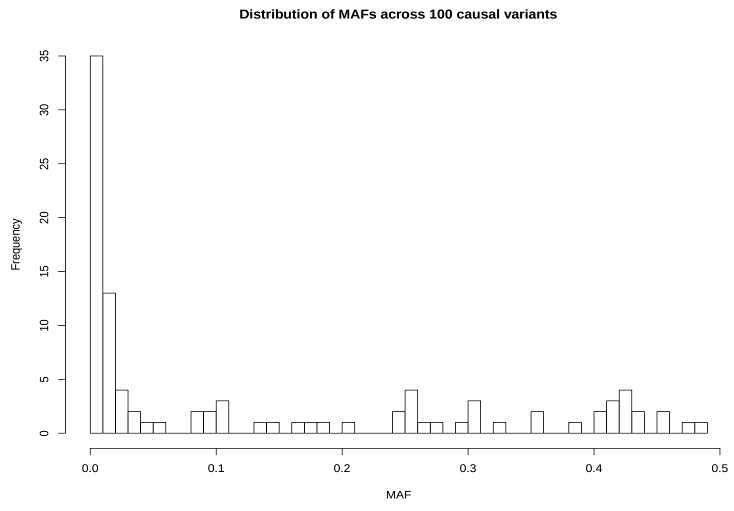

Figure 5: Distribution of MAFs across 100 causal SNPs

simulations, we chose 10 genes at random, and among each of them 10 SNPs at random, for a total of 100 causal SNPs, each assigned to have a relative risk of 2.0. Figure 5 shows the distribution among these causal SNPs. The small relative risk ensured that rare causal SNPs would be difficult to detect unless prior biological knowledge (i.e. gene annotations) could facilitate their inclusion into the model through their L2 norm. The receiver operating curve was plotted by estimating the true and false discovery rates for both a mixed penalty (i.e. $\lambda_L = \lambda_E$) and a pure L1 penalty, taken across 100 simulation replicates. We did not extensively explore a full range of L1 to L2 ratios in this simulation. Rather, we simply chose parameter values based on ROCs from simulations in [Zhou *et al.*, 2010], suggesting that a λ_L to full penalty (i.e. $\lambda_L + \lambda_E$) ratio of .5 yields superior power compared to pure L1 or L2 based penalties at low FDR levels.

1.5 Stability Selection for GWAS

Stability selection was developed by Meinhausen and Bühlmann to address the issue of model overfitting in the setting of L1 penalized regression [Meinshausen and Bühlmann, 2010]. A models is fit to each random subsample (generally half) of the entire dataset over a set Λ of penalty values λ_L. Variables are declared "stable" if the proportion of subsamples containing the variable is at least as large as a threshold, or formally:

$$\hat{S}^{stable} = \{k : \max_{\lambda_L \in \Lambda} \hat{\Pi}_k^{\lambda_L} \geq \pi_{thr}\} \tag{7}$$

One important feature of stability selection is the ability to provide error control. The expected number V of

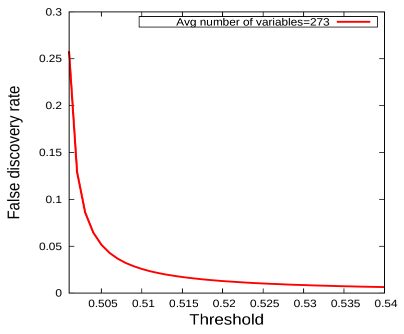

Figure 6: False discovery rate as a function of stability selection threshold

falsely selected variables is:

$$E(V) \leq \frac{1}{2\pi_{thr} - 1} \frac{q_\Lambda^2}{p} \tag{8}$$

where q_Λ is the average number of selected variables taken across all replicates, over the set Λ. Determining the set of stable variables according to Eq 7 can pose a heavy computational burden since at each replicate, each model is fit across multiple values of λ_L. This leads to impractical run times, even with parallel implementations. The authors point out however, for methods such as the LASSO where the probability of inclusion is a monotonic function with respect to decreasing values of λ_L, it is sufficient to choose a small enough value of λ'_L such that the true set of variables $\hat{S}^{\lambda_L}$ under the true value of λ_L is contained within the larger set $\hat{S}^{\lambda'_L}$ with high probability. In other words, by allowing moderate overfitting, we can apply the method as defined in Eq 7 where the set Λ contains only value λ'_L.

For the African American GWAS data, each replicate's value of λ'_L was set at 100, which resulted in 273 variables being selected on average (i.e. q_Λ=273). Note that because we did not include L2 penalties, λ_E is set to zero. Based on λ'_L=100, one can get a sense of error control as a function of the threshold π_{thr}, as shown in Figure 6.

1.6 Framework Use

We now walk the reader through source code for a program that makes use of our C++ framework. The **Stability** class, which implements Stability Selection, is a sub-class of the **MpiLasso2** class, which itself handles details such as loading XML configuration files, parsing standardized format data files (e.g. PLINK files [Purcell *et al.*, 2007]), communicating amongst MPI nodes, and executing OpenCL kernels on GPUs. One can refer to the code for **Power.cpp**, which implements the rare variant analysis, at the Google Code site.

The header file below defines a container called **stability_settings_t** that is an extension of the XML container defined in **MpiLasso2.hpp**. Configuration settings that are relevant to the analysis can be defined by the user here, mapping to corresponding XML tags in the configuration file.

Stability.hpp

```
class stability_settings_t:public lasso2_settings_t{
public:
  stability_settings_t(const ptree & pt,int mpi_rank);
  int replicates;
  string mask_basepath;
};

class Stability:public MpiLasso2{
public:
  Stability();
  ~Stability();
  void init(const ptree & pt);
  void run();
private:
  int * varcounts;
  int totaltasks;
  vector<string> varnames;
  int * mask;
};
```

The C++ class listed below initializes analysis by loading configuration settings, PLINK formatted data files, and other ancillary files. The initialization routine also performs the important tasks of initializing all MPI data structures, GPU memory buffers, and transfer of partitioned datasets to the computing nodes. The actual execution of the Stability Selection algorithm carries out a loop which reads a mask file at each iteration that informs the optimizer of which subjects to include for that iteration. The routine concludes by writing to the output file selection probabilities for a given value of λ_L.

Stability.cpp

```
#include<cstdlib>
#include<cstring>
#include<iostream>
#include<sstream>
#include<fstream>
#include<math.h>
#ifdef USE_GPU
#include<CL/cl.hpp>
```

```
#include"clsafe.h"
#endif
#include"main.hpp"
#include"analyzer.hpp"
#include"io.hpp"
#include"dimension2.h"
#include"utility.hpp"
#include"lasso_mpi2.hpp"
#include"stability.hpp"

using namespace std;
typedef unsigned int uint;

// extends the XML configuration file container for LASSO settings
// specific to Stability Selection

stability_settings_t::stability_settings_t(const ptree & pt,int mpi_rank):
lasso2_settings_t(pt,mpi_rank){
  replicates = pt.get<int>("subsamples");
  mask_basepath = pt.get<string>("inputdata.mask_basepath");
}

Stability::Stability():MpiLasso2(){}

Stability::~Stability(){
  delete settings;
}

void Stability::init(const ptree & pt){
  // For loading configuration settings
  settings = new stability_settings_t(pt,this->get_rank());
  ofs<<"Platform ID: "<<settings->platform_id<<endl;
  ofs<<"Device ID: "<<settings->device_id<<endl;
  ofs<<"Kernel Path: "<<settings->kernel_path<<endl;
  // Read data files
  read_data(settings->snpfile.data(),settings->pedfile.data(),
  settings->genofile.data(),settings->covariatedatafile.data(),
  settings->covariateselectionfile.data(),NULL);
  read_tasks(settings->tasklist.data(),settings->annotationfile.data());
  varnames = get_tasknames();
  totaltasks = varnames.size();
  varcounts = new int[totaltasks];
  ofs<<"There are "<<totaltasks<<" total tasks\n";
  allocate_datastructures();
  send_phenotypes();
  send_covariates();
  send_genotypes();
  send_tuning_params();
}

void Stability::run(){
  stability_settings_t * settings =
  static_cast<stability_settings_t * >(this->settings);
  for(int i=0;i<totaltasks;++i) varcounts[i] = 0;
  for(int replicate = 0;replicate<settings->replicates;++replicate){
    ostringstream oss;
    oss<<settings->mask_basepath<<"."<<replicate;
    read_data(NULL,NULL,NULL,NULL,NULL,oss.str().data());
    send_mask();
    double logL;
    vector<modelvariable_t> modelvariables;
    fitLassoGreedy(replicate, logL,modelvariables);
    if (is_master){
      ofs<<"REPLICATE:\t"<<replicate<<endl;
      ofs<<"LAMBDA:\t"<<settings->lambda<<endl;
      int modelsize = modelvariables.size();
      for(int i=0;i<modelsize;++i){
        ++varcounts[modelvariables[i].index];
      }
    }
  }
  if(is_master){
    for(int i=0;i<totaltasks;++i){
      ofs<<"Selection probabilities:\n";
      if (varcounts[i]>0){
        ofs<<varnames[i]<<":\t"<<1.*varcounts[i]/settings->replicates<<endl;
      }
    }
  }
  cleanup();
}
```

References

DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., Hanna, M., McKenna, A., Fennell, T. J., Kernytsky, A. M., Sivachenko, A. Y., Cibulskis, K., Gabriel, S. B., Altshuler, D., and Daly, M. J. (2011). A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet.*, **43**, 491–498.

Khronos (2011). Khronos group. `http://www.khronos.org`.

Meinshausen, N. and Bühlmann, P. (2010). Stability selection. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **72**(4), 417–473.

nVidia (2011). nvidia cuda. `http://www.nvidia.com/object/cuda_home_new.html`.

OpenMP (2011). Openmp. `http://openmp.org/wp/`.

Purcell, S., Neale, B., Todd-Brown, K., Thomas, L., Ferreira, M. A., Bender, D., Maller, J., Sklar, P., de Bakker, P. I., Daly, M. J., and Sham, P. C. (2007). PLINK: a tool set for whole-genome association and population-based linkage analyses. *Am. J. Hum. Genet.*, **81**, 559–575.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, **58**(1), pp. 267–288.

Wu, T. T. and Lange, K. (2008). Coordinate descent algorithms for lasso penalized regression. *Annals of Applied Statistics*, **2**, 224–244.

Zhang, T. and J. Oles, F. (2001). Text categorization based on regularized linear classification methods. *Inf. Retr.*, **4**, 5–31.

Zhou, H. *et al.* (2010). Association screening of common and rare genetic variants by penalized regression. *Bioinformatics*, **26**, 2375–2382.

Ziegler, A., MacCluer, J. W., and Almasy, A. (2011). Genetic Analysis Workshop 17: Approaches to Analysis of Next-Generation Sequencing Data. *Genetic Epidemiology (in press)*, **35**(8).

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **67**(2), 301–320.