33rd Annual International Conference of the IEEE EMBS
Boston, Massachusetts USA, August 30 - September 3, 2011

An Evolutionary Optimization Strategy Using Graphics Processing Units to Efficiently Investigate Gene-Gene Interactions in Genetic Association Studies

Joel B. Fontanarosa and Yang Dai, *Member, IEEE*

Abstract— **The analysis of gene-gene interactions related to common complex human diseases is complicated by the increasing scale of genetic association analysis. Concurrent with the advances in genetic technology that led to these large data sets, improvements have been made in parallel computing with graphics processing units (GPUs). The data-intensive nature of genetic association analysis makes this problem particularly suitable for improved computation with the powerful computing resources available in GPUs. In this study, we present a GPU-accelerated discrete optimization strategy to improve the computational efficiency of multi-locus association analysis. We implemented an adaptive evolutionary algorithm that takes advantage of linkage disequilibrium to reduce the need for exhaustive search for combinations of genetic markers. The proposed GPU algorithm was shown to have improved efficiency and equivalent power relative to the CPU version.**

I. Introduction

HIGH density genetic association studies have been conducted for dozens of common complex diseases, providing geneticists with a wealth of information about the underlying genotypic features in a variety of populations. These studies have led to the discovery of numerous genetic variants shown to be reliably associated with specific disease phenotypes. However, as researchers seek to better understand genetic diseases by using analyses of gene-gene interactions, by collecting denser genetic association data, or by conducting meta-analyses of large combined data sets, the scale and complexity of the computations involved in the analysis increase dramatically [1, 2].

To address this growing computational burden, we present a parallel evolutionary optimization strategy that uses local linkage disequilibrium structure and graphics computing units (GPUs) to improve the power to search for gene-gene interactions. Several research groups have presented impressive results using GPUs to improve the computational speed of analyses for minimal cost [3-5]. These methods have significantly reduced the amount of time required to analyze interactions in large genetic association studies. Our parallel algorithm expands on the recent papers describing GPU-accelerated gene-gene interaction analysis to take advantage of the previously demonstrated improvement in algorithmic performance in a powerful evolutionary optimization framework [6]. We demonstrate the computational speed of our method using an exhaustive analysis of a moderately sized simulation data set, and show that the power of our GPU-accelerated implementation to detect causal interactions is equivalent to that of our previously published method [6].

Manuscript submitted April 15, 2011.

JF and YD are with the Bioinformatics Program in the Department of Bioengineering at The University of Illinois at Chicago, Chicago, IL 60607 USA. (Corresponding author: JF, phone: 312-413-0191; e-mail: jfonta3@uic.edu).

II. Methods

A. Proposed algorithm and GPU implementation

In our previous work, we proposed a block-based evolutionary optimization procedure [6] for investigating gene-gene interactions in genome-wide case-control studies. Using linkage information to divide a genetic association study into discrete, highly correlated blocks can improve the power and computational efficiency of a search for gene-gene interactions. A population, i.e., a set of candidate solutions, defined as the combinations of k-blocks of genotypes, is randomly built up and then is improved iteratively with respect to some predefined fitness function (e.g. χ^2 statistic associated with a k-block genotype combination). The solutions are modified by introducing random new combinations (immigrants), by modifying part of an existing combination (mutations), or by combining solutions within the set (recombination), and the solutions with the highest fitness are selected. The procedure continues iteratively until a stable set of gene-gene combinations with the best fitness measures results.

Our implementation used this general optimization procedure to determine the linkage disequilibrium block combinations with the best fitness. We define the fitness measure for each combination of blocks $B_{s1}, B_{s2}\ldots B_{sk}$ as described previously [6]. Let S be the set of all possible k-SNP genotype combinations with exactly one SNP from each linkage block. The genotypes in each B_{sm} ($1 \leq m \leq k$) all have 3 possible genotypes, and the observed frequency counts for the 3^k possible genotypes for each genotype combination are tallied in a 2×3^k contingency table G_s of case and control genotype counts. For each s in S with contingency table G_s, we consider the χ^2 statistic with $3^k - 1$ degrees of freedom. The fitness of each block combination is then determined as:

$\varphi = \max\{\chi^2_{3^{k-1}}(G_s) : s \in S\}$. The maximum search depth of S in each iteration is set as a parameter. When k is small (≤ 3), a deep search for the optimal φ from the set S is used. For larger k, φ is determined by random sampling of S.

The main computational burden in our evolutionary optimization algorithm is the calculation of genotype frequency counts for k-SNP combinations. Tallying the genotype counts for a set of k-SNP combinations in a large population of individuals can be computed very efficiently on a GPU using a modified parallel reduction algorithm (a toy example for k=2 for 4 subjects is shown in Fig. 1) [4, 7]. We used the NVIDIA Compute Unified Device Architecture (CUDA) for our GPU implementation. The CUDA programming model is a single-instruction, multiple-data platform that executes functions using parallel *threads* within units of data grouped into *GPU blocks*. Multiple GPU blocks can then be run in parallel using the GPU multiprocessors. Our implementation uses two main memory spaces on the GPU: (1) the larger (and slower) *global memory* for genotype storage, and (2) the smaller (and faster) *shared memory* for tallying frequency counts. Global memory is the largest memory space on the GPU, can be accessed by any GPU block, and is the only data directly accessible from the CPU. By contrast, shared memory is much smaller, and is only accessible by threads within a single GPU block.

As described in previous implementations of GPU gene-gene interaction analysis, a common way to achieve optimal performance is to structure the data such that the load is balanced across the resources of the GPU, with the majority of data-operations occurring in the fastest performing shared memory cache [4, 5, 7]. Genotypes are copied onto the GPU in global memory. For a study with N patients, each k-SNP combination is then copied into a $N*3^k$ element vector in shared memory and reduced to a list of frequency counts for each of the possible 3^k genotypes (Fig. 1). If a vector of $N*3^k$ elements does not fit within a single GPU block, the frequency counts are split into multiple GPU blocks, and the frequencies for all N patients are combined in global memory after the reduction. To maximally take advantage of the GPU resources and to minimize overhead due to data transfer between the GPU and the CPU, we define a parameter X as the number of k-SNP combinations calculated in parallel on the GPU during each iteration. The value of X depends on the specific GPU being used, and is based on the amount of GPU memory, the number of GPU multiprocessors, the maximum number of GPU blocks that may be run concurrently on the GPU, and the number of blocks used per frequency count (based on N and the amount of GPU shared memory).

The evolutionary population size (P) that was determined to be the best for exploring genetic interactions in our previous study is much smaller than X. To adapt our evolutionary optimization framework to a parallel version that maximally takes advantage of the GPU resources while maintaining power to detect causal k-SNP interactions, we used a parallel islands approach (Fig. 2). In this method, we initialize a set of F separate populations ("islands"), each of which has P solutions, such that there are a total of X k-SNP combinations across all islands. Each island population is then modified using immigration, mutation, and recombination for a finite number of iterations. The probabilities for the moves in the evolutionary algorithm were defined as follows:

1) *Immigration*: p=0.6, randomly select a new block combination
2) Mutation: p=0.2
    a) Substitute one of the blocks with a random block
    b) Modify one of the two blocks by randomly choosing a nearby block
3) *Recombination*: p=0.2, substitute one of the blocks

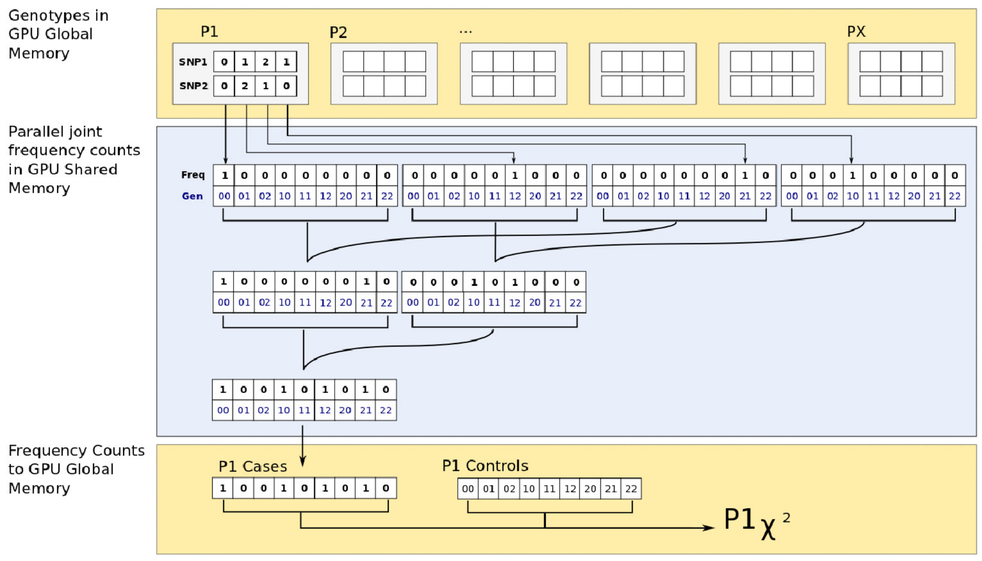
Genotypes in GPU Global Memory
P1
P2
...
PX
SNP1
SNP2
Parallel joint frequency counts in GPU Shared Memory
Freq
Gen
Frequency Counts to GPU Global Memory
P1 Cases
P1 Controls
P1 χ²

Fig. 1. Overview of the parallel reduction approach. In this example, we show population k-SNP (k=2) combinations P1 through PX split up into X GPU blocks. We show an example parallel reduction for P1 with a two-SNP combination for 4 subjects in the case group. These genotype frequency counts are copied from the GPU global memory to shared memory to rapidly tally the frequency counts in parallel for all X GPU blocks. The frequency counts are then copied back to the GPU global memory, where they can be used to calculate the fitness measure before being copied to the CPU.

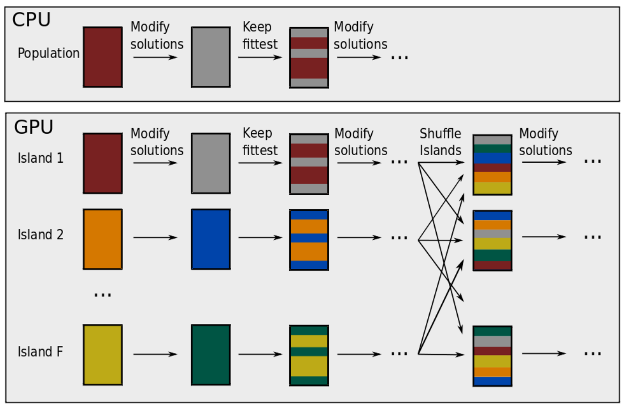

Fig. 2. Schematic comparing the CPU Evolutionary Algorithm with the GPU Parallel Islands implementation. In our CPU procedure, an initial "population" of combinations is iteratively modified using the immigration, mutation, and recombination moves described in the text. This process continues until a maximum number of iterations is reached or a convergence criterion is satisfied. In the GPU implementation, we initialize a set of F "island" populations. Each island is a population of combinations that is independently modified using the evolutionary moves. After a set number of iterations, the combinations across all islands are shuffled and the process repeats until the convergence criteria is satisfied.

with a block from another randomly selected solution in the population

The size of the population of solutions, P, is a parameter that was set at 100. We defined two termination criteria for our algorithm: (1) a limit on the number of total iterations (default = M/10, where M is the number of SNPs) or (2) a limit on the number of iterations that may pass without any change in the set of fittest solutions. To ensure a proper comparison, the evolutionary moves, the move probabilities, and the definition of φ were the identical in the CPU and GPU implementations.

In the GPU implementation, the island populations are independently optimized for a set number of iterations until the fitness of the solutions in each island is improved. The combinations across all islands are then shuffled together and regrouped into F populations, and the procedure is repeated until a stable set of gene-gene combinations results.

The number of independent islands (F) is a parameter that was defined as a function of the number of GPUs, X, and the number of fitness measures to calculate in each population. This approach allows our model to be adjusted to fit the resources of the specific GPU being used in order to maximize performance. We determined the performance of our code using a computer with an i5 Quad-core 2.66GHz processor, 8GB of RAM and a NVIDIA GTX 470 GPU running CUDA 3.2 on Ubuntu Linux 10.10. GPU code was implemented and tested in CUDA or C for best performance and then wrapped with Python 2.6 using the PyCUDA library (2011.1) [8].

B. Simulation model and performance analysis

To establish equivalence in statistical power of our GPU implementation with that of the CPU implementation of our evolutionary optimization strategy for moderately sized studies, simulation models were built as described previously [6, 9] using a multiplicative two-locus disease model. In each simulation data set of 5000 SNPs, the disease prevalence was fixed at 0.01, the genotypic effect size was set at a relative risk of ω=1.25, the Minor Allele Frequency (MAF) was set at 0.15 or 0.30, and the population was set to 1000, 2000, and 5000. The disease model included two loci: each with a marginal effect and an effect size of ω*ω for any interaction between disease alleles at the two causal loci.

Because the running time of our optimization procedure is dependent on several parameters and random chance, we used exhaustive analyses over smaller sets of SNPs on random simulation data (no simulated effect) to measure the performance of our code: N = (2000, 5000, 10000), M = (1000, 2000, 5000), where N is the number of samples and M the number of SNPs.

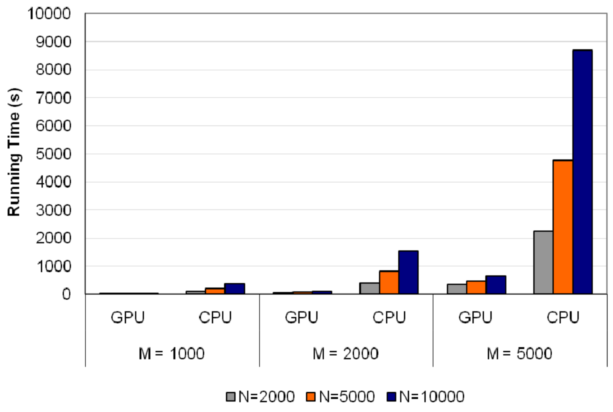

Fig. 3. Comparison of GPU and CPU running times in seconds for exhaustive search on data sets of varying size. N is the number of subjects and M is the number of SNPs.

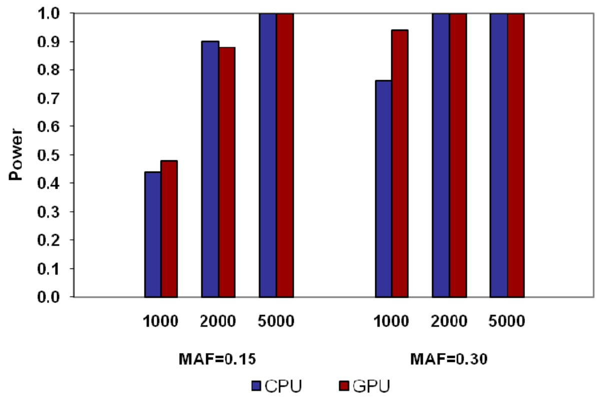

Fig. 4. Comparison of the statistical power of the GPU and CPU implementations to detect causal genetic markers. Power is reported as the proportion of times the causal gene-gene interaction was found in 50 simulations for M=5000,and N=(1000,2000,5000).

The population and linkage disequilibrium features in our simulation data were based on HapMap CEU phased data (CEPH samples with ancestry from Northern and Western Europe) for autosomal SNPs corresponding to the Illumina HumanHap 550K SNP array [9, 10]. Linkage structure was measured and included in our models as described previously [6].

III. Results

Since the running time of our optimization procedure may be variable, we elected to use a limited exhaustive search to assess the comparative running time of our GPU and CPU codes. The sample sizes N=(2000, 5000, and 10000) correspond roughly with a moderately sized GWAS, a large GWAS [11], and a combined analysis of more than one GWAS. As shown in Fig. 3, the running times are comparable for small studies, but the computational benefit of the parallel implementation becomes readily evident for exhaustive analyses of even a limited set of SNPs.

It was important to establish that the GPU and CPU implementations of our optimization approach had equivalent power to detect a simulated gene-gene interaction. We defined power as the proportion of times out of 50 replicate simulation data sets that the causal interaction was detected. As shown in Fig. 4, the GPU and CPU versions of our optimization algorithm have nearly identical power.

IV. Discussion

As shown in previous studies [3-5], parallel GPU computing techniques can substantially improve the performance of genetic analysis tools. Our previously published block-based evolutionary optimization strategy was designed to improve the power and efficiency of gene-gene interaction detection by taking advantage of local linkage disequilibrium structure. The parallel implementation of our algorithm extends this approach so that it can be more conveniently applied for exploratory analysis of genome-scale data. We demonstrated that our GPU method has equivalent statistical power using a standard multiplicative two-locus disease model that included marginal effects. As has been shown previously, this power is expected to be higher for our method when the sample size is large and when the causal allele is more common [6, 12]. Any differences in power between the two implementations (as shown in Fig. 4) are due to random variations in our evolutionary algorithm or a slightly increased search space of the GPU implementation of the algorithm.

One limitation of our current software is that it will only run on devices compatible with NVIDIA CUDA. Forthcoming versions of our code will allow our software to be run on hardware from other vendors by translating the necessary GPU functions from CUDA to OpenCL [13]. In our future research, we plan to expand the set of genomic association analysis tools accelerated by our parallel implementation, and we will also explore new ways of further improving the performance of our code as new hardware and techniques emerge.

References

[1] H. J. Cordell, "Detecting gene-gene interactions that underlie human diseases," *Nat Rev Genet,* vol. 10, pp. 392-404, Jun 2009.

[2] E. E. Eichler, *et al.*, "Missing heritability and strategies for finding the underlying causes of complex disease," *Nat Rev Genet,* vol. 11, pp. 446-50, Jun 2010.

[3] X. Hu, *et al.*, "SHEsisEpi, a GPU-enhanced genome-wide SNP-SNP interaction scanning algorithm, efficiently reveals the risk genetic epistasis in bipolar disorder," *Cell Res,* vol. 20, pp. 854-7, Jul 2010.

[4] N. A. Sinnott-Armstrong, *et al.*, "Accelerating epistasis analysis in human genetics with consumer graphics hardware," *BMC Res Notes,* vol. 2, p. 149, 2009.

[5] L. S. Yung, *et al.*, "GBOOST : A GPU-based tool for detecting gene-gene interactions in genome-wide case control studies," *Bioinformatics,* Mar 3 2011.

[6] J. Fontanarosa and Y. Dai, "A block-based evolutionary optimization strategy to investigate gene-gene interactions in genetic association studies," *Proceeding of 2010 IEEE International conference on Bioinformatics and Biomedicine Workshop,* pp. 330-335, 2010.

[7] M. Harris. (2009). *Optimizing Parallel Reduction in CUDA*. Available: developer.download.nvidia.com

[8] A. Klöckner, *et al.*, "PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation," *arXiv,* 2011.

[9] C. Li and M. Li, "GWAsimulator: a rapid whole-genome simulation program," *Bioinformatics,* vol. 24, pp. 140-2, Jan 1 2008.

[10] K. A. Frazer, *et al.*, "A second generation human haplotype map of over 3.1 million SNPs," *Nature,* vol. 449, pp. 851-61, Oct 18 2007.

[11] "Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls," *Nature,* vol. 447, pp. 661-78, Jun 7 2007.

[12] J. Marchini, *et al.*, "Genome-wide strategies for detecting multiple loci that influence complex diseases," *Nat Genet,* vol. 37, pp. 413-7, Apr 2005.

[13] Khronos. (2011). *OpenCL - The open standard for parallel programming of heterogeneous systems*. Available: http://www.khronos.org/opencl/