Quantitative Trait Locus Analysis Using a Partitioned Linear Model on a GPU Cluster

Peter E. Bailey*, Tapasya Patki*, Gregory M. Striemer†, Ali Akoglu†, David K. Lowenthal*, Peter Bradbury‡
Matthew Vaughn§, Liya Wang¶, Stephen Goff‖
*Department of Computer Science, The University of Arizona
†Department of Electrical and Computer Engineering, The University of Arizona
‡USDA/ARS and Cornell University
§Texas Advanced Computing Center
¶Cold Spring Harbor Laboratory
‖iPlant Collaborative

***Abstract*—Quantitative Trait Locus (QTL) analysis is a statistical technique that allows understanding of the relationship between plant genotypes and the resultant continuous phenotypes in non-constant environments. This requires generation and processing of large datasets, which makes analysis challenging and slow. One approach, which is the subject of this paper, is Partitioned Linear Modeling (PLM), lends itself well to parallelization, both by MPI between nodes and by GPU within nodes. Large input datasets make this parallelization on the GPU non-trivial.**

This paper compares several candidate integrated MPI/GPU parallel implementations of PLM on a cluster of GPUs for varied data sets. We compare them to a naive implementation and show that while that implementation is quite efficient on small data sets, when the data set is large, data-transfer overhead dominates an all-GPU implementation of PLM. We show that an MPI implementation that selectively uses the GPU for a relative minority of the code performs best and results in a 64% improvement over the MPI/CPU version. As a first implementation of PLM on GPUs, our work serves as a reminder that different GPU implementations are needed, depending on the size of the working set, and that data intensive applications are not necessarily trivially parallelizable with GPUs.

I. Introduction

According to the National Research Council, one of the foremost challenges in plant biology is to predict the collection of traits (i.e. phenotype) for a particular plant, in a non-constant environment, given a readout of its genetic code. Being able to predict altered plant responses in an environment that is undergoing anthropogenic change is very important for conducting research in domains like plant development and adaptation, plant physiology, crop improvement and ecological genomics [1].

Genetic traits can be qualitative (discrete) or quantitative (continuous). Quantitative traits depend on more than one gene and are influenced by the environment. The genetic architecture of diverse phenotypes can be determined with the help of a statistical technique called Quantitative Trait Locus (QTL) analysis. QTL analysis can be challenging as it requires the generation and processing of large amounts of data. A Partitioned Linear Model (PLM) that is amenable to large-scale parallelism can be developed to perform QTL analysis [2].

This paper focuses on comparing implementations of PLM with both small and large input data sets. We first created an MPI application and then turned to investigating how amenable PLM is to GPU parallelization within each MPI process. The data set size is quite important in how best to design an implementation, because for large sizes, the benefit of GPU parallelization can be outweighed by the overhead to transfer the data from the CPU to the GPU. We found that indeed, a different implementation is required for the small and large data sets. For small input sets, a typical GPU implementation where the GPU performs all the computation achieves speedup; but on large data sets, that same GPU implementation actually caused *slowdown*. For the large input set, we then investigated multiple implementations that selectively parallelize only small portions of the MPI process code onto the GPU. We achieved improvement up to 64% over the MPI-only implementation. While the GPU parallelized portion performs quite well, the portion of the MPI process code that is parallelized is relatively small.

The lesson learned is that GPU parallelization, even when significant programmer time is invested, does not necessarily result in a several-fold improvement (or a several-hundred-fold improvement, as is often reported in the GPU literature). Moreover, there is not a "one size fits all" GPU implementation for a given problem; the best implementation may be input dependent—it is in our application, as it depends on whether the input data set fits completely in GPU

RAM.

The rest of this paper is organized as follows. Section II discusses the PLM problem. Section III describes our implementation, and Section IV provides performance results. Section V discusses work related to this paper, and Section VI summarizes the paper.

II. Overview

The Plant Science Cyberinfrastructure Collaborative (PSCIC) program is intended by NSF to create a new organization—a cyberinfrastructure collaborative for the plant sciences—that will enable conceptual advances through integrative, computational thinking [3]. The *iPlant Collaborative* (iPC) [4] involves using modern computational science and cyberinfrastructure solutions to address grand challenges in the plant sciences. iPG2P—relating plant genotypes to phenotypes in complex environments—is one of the current projects of the iPC. This initiative brings together researchers and experts from various domains like plant biology, bioinformatics, computational genetics, statistics and computer science.

The problem we are addressing is stated as follows. We are given (1) a particular species of plant (e.g. maize, rice, soybean), (2) a genetic description of an individual (genotype), and (3) a phenotype, the trait of interest, (flowering time, yield, or any of hundreds of others). The goal is to predict, in non-constant environments, the quantitative result (phenotype) that causes the desired trait in different subspecies.

A. QTL Analysis

As discussed in Section I, QTL analysis is a statistical method that allows us to explain the genetic basis of variation in complex traits. Scientists require two pieces of data to perform such an analysis: a set of genotypes and at least one phenotype for each group of individuals drawn from a population of interest. Based on a regression analysis, the traits occurring due to a given phenotype can be determined and traced across the group of individuals in the species. If the group of individuals is sufficiently large and properly samples the population, the results should apply to the population as a whole.

As an example, consider a set of maize inbred lines from a broad range of backgrounds that vary in the number of days from planting to pollen shed. These can be chosen to study the genes affecting flowering time and represents a phenotype data set that could be used to make inferences about maize in general. The genotype data might be a large set of genetic markers that distinguish between these lines scored on the same set of individuals.

A single nucleotide polymorphism, or SNP, can be defined as single base pair within a DNA sequence that can differ among individuals and can lead to genetic variation. An example of a SNP is the change from A to T in the sequences AATGCT and ATTGCT. SNPs serve as genetic markers for QTL as they are mutations that have been successful in surviving and occur in a significant proportion of the population of a species.

SNPs can be used to divide the individuals into two classes. Testing whether the two classes are statistically different for the phenotype of interest is straightforward. After applying this test to all the SNPs in the genotype set, the SNP that results in classes with the highest probability of being different is selected. That SNP is added to the linear model, the fixed effects matrix, and all other SNPs are retested in iterations to locate a second, third, fourth, etc. SNPs. Through a series of these steps, a forward regression model is built.

B. Forward Regression using PLM

In this work we use PLM for forward regression due to its potential for parallelization. Figure 1 illustrates the workflow for forward regression, and Algorithm 1 provides pseudocode for it. The algorithm iterates through each possible input SNP to calculate an F-value (see Algorithm 1). The SNP with the highest F-value is selected. Each SNP contributes to a column of the genotype set in Algorithm 1. The input fixed effects matrix, *M*, contains a number of rows equal to the length of the input SNPs, or *lenSNP*. Initially each of the columns of *M* represent each of the populations. Each row is a genotyped homozygous inbred individual plant. A two-valued indicator variable is used for SNP values (0 or 2). Floating point data is the imputed genotype based on flanking markers and weighted by their genetic distances [5].

An advantage of PLM is that it is amenable to large-scale parallelism on the GPU; it exhibits data-parallelism at the SNP level. Computations performed on a SNP are independent of the ones being performed on other SNPs. Steps 5 to 14 of Algorithm 1 indicate the set of operations that the kernel performs on each SNP independently. At the end of each iteration, a reduction to determine the maximum F-value is carried out, followed by updating the fixed-effects matrix with the chosen SNP. This process is also suitable for mapping onto a GPU.

When using forward regression, PLM can be distinguished from a similar linear model called the General Linear Model (GLM) by the way it calculates $beta$ in Step 6. GLM uses a $O(numCols^3)$ algorithm

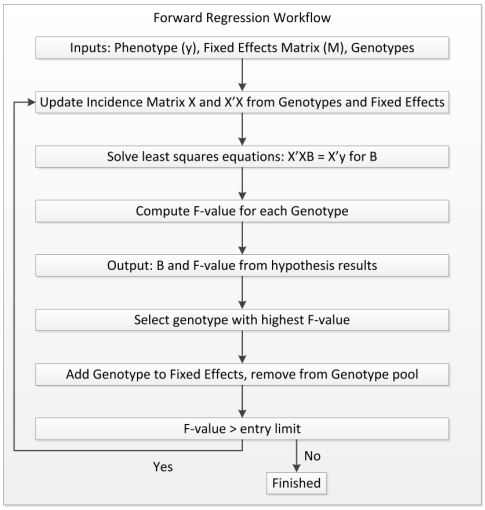

Figure 1. Forward regression workflow

to calculate the inverse, where $numCols$ is the number of columns in the X matrix. This is followed by a matrix-matrix multiplication to calculate $beta$ in each iteration. PLM, on the other hand, calculates the inverse and $X^T y$ incrementally at the end of each iteration, which reduces the complexity to $O(numCols^2)$. Note that if we have simple regression (with only one iteration), both these algorithms would be equally efficient. When we add a new column (say SNP_j) to the X matrix (Step 5), we can calculate the new $X^T X$ as follows:

$$X_{new}^T X_{new} = \begin{bmatrix} X^T X & X^T SNP_j \\ SNP_j^T X & SNP_j^T SNP_j \end{bmatrix}$$

Here, X_{new} is the matrix X appended with SNP_j, which is the SNP under consideration. Updating $(X_{new}^T X_{new})^{-1}$ given a partitioned $X_{new}^T X_{new}$ has been discussed by Hager [6]. Similarly, the new $X^T y$ can be calculated incrementally with the help of the old $X^T y$ and $SNP^T y$.

III. Implementation

Our parallel implementation of PLM is hierarchical: the top-level parallelization (across nodes) is via MPI, and then each MPI process uses GPU parallelization. We describe each in turn in this section.

A. Inter-Node Parallelization with MPI

We use MPI to parallelize our implementation of forward regression with PLM. All MPI ranks are statically-assigned, equally-sized, disjoint portions of the input SNP data, with each rank receiving an integral number of SNPs. All ranks retain the fixed effects and residuals in their entirety.

Once the work has been divided, the following process continues until iteration termination: a single iteration of forward regression continues locally on each rank until a local maximum F value is found. A max-reduction is then performed across all ranks to find the global maximum F value, F_{max}. The SNP associated with F_{max} is then broadcast to all ranks, which each rank uses to update its local model in preparation for the next iteration.

After all iterations are complete, the indices of SNPs incorporated into the model and the associated P values are reduced from their originating ranks onto rank 0, where they are written to a file.

B. General Intra-Node GPU Parallelization

As PLM is at the core of forward regression, we first detail our implementation of a PLM kernel. The kernel takes as inputs the intermediate terms $X^T SNP$ and $SNP^T SNP$ for each SNP, and $G = (X^T X)^{-1}$, y, and $errorSS$ globally, with output F for each SNP.

After the calculation of $X^T SNP$, most of the computational work in PLM is due to the calculation of $G * X^T SNP$. Thus, we optimize the design of the PLM kernel to efficiently compute $G * X^T SNP$ using $numCols$ threads, where $numCols$ is the number of columns in the design matrix, X.

This mapping of SNPs to thread blocks is comparatively efficient, considering its alternatives. If we were to use a single thread per SNP, each thread would be forced to make more reads from global memory due to lack of sufficient storage in registers or shared memory for intermediate terms. While this arrangement would allow for greater multiprocessor occupancy via larger thread blocks, the additional memory-induced latency would not be worthwhile, especially given the low computational intensity of the problem.

Similarly, a mapping of multiple SNPs per thread block is unlikely to improve performance, as it would require either additional registers per thread if the SNPs in a block were processed serially, or additional threads per block and additional registers per thread if the SNPs in block were processed in parallel. Both cases are likely to have limited occupancy due to register usage, and the latter case will suffer from warp divergence if not implemented carefully.

The remaining work consists of vector-vector and matrix-vector operations between $numCols$-unit vectors and the $numCols$-by-$numCols$ matrix $G = (X^T X)^{-1}$. If we were to use $numCols^2$ threads per SNP, computation of the matrix-vector product would be inefficient, because the occupancy would be limited to a single SNP per multiprocessor. With

Algorithm 1

PLM pseudocode. The inputs are (1) a genotype set, *SNP*, comprising of *n* SNPs, each with length *lenSNP*; (2) a fixed effects matrix, *M*, with *lenSNP* rows, which initially contains 26 columns; and (3) the phenotype, *y*, which is a column vector of length *lenSNP*. The algorithm iteration threshold is determined by *iter*.

1: $TotalSS \leftarrow y^T y$
2: $K \leftarrow$ *[zeros(1:n-1) 1]*
3: **for** $i = 0 \rightarrow iter$ **do**
4: **for** $j = 0 \rightarrow n$ **do**
5: $X \leftarrow append(M, SNP[j])$
6: $beta \leftarrow (X^T X)^{-1} * (X^T y)$ ▷ Calculated Incrementally
7: $ModelSS \leftarrow beta^T * X^T y$
8: $TotalSS \leftarrow ErrorSS - ModelSS$
9: $F = [(K^T * beta)^T \times (K^T GK)^{-1} \times (K^T * beta)]/[ErrorSS/(n - rank(X))]$
10: **if** $F_{temp} > threshold$ **then**
11: $Fvalue[j] \leftarrow F_{temp}$
12: **else**
13: $Fvalue[j] \leftarrow 0$
14: $return$
15: **end if**
16: **end for**
17: $F_{max} \leftarrow max(Fvalue[1:n])$
18: $genotype \leftarrow$ *SNP with* F_{max}
19: $M \leftarrow append(M, genotype)$
20: **end for**

$numCols$ threads per SNP, occupancy reaches eight SNPs per multiprocessor.

C. Intra-Node GPU Parallelization on Small Datasets

If the input problem is small enough that all SNP data fits entirely in GPU RAM on each MPI rank, we use the CPU primarily for MPI communication between iterations. All computation, with the exception of updating the model with the F_{max} SNP, takes place on the GPU. Using this approach, we minimize overhead associated with data transfer. Additionally, we benefit from the relatively high memory bandwidth of the GPU.

In addition to the PLM kernel, the implementation for small datasets also features a kernel to compute the dot product of each pair of SNPs in parallel. This kernel efficiently computes the dot product of each column of a matrix with itself. The kernel computes one column dot product per thread block. Computing a single dot product per thread is possible, but our approach is more appropriate for column-major matrices, which follow the CUBLAS data storage convention.

D. Intra-Node GPU Parallelization on Large Datasets

For large problems, the SNP data does not fit into GPU RAM, which presents issues that must be addressed. Two approaches are possible. In the first approach, all computation is performed on the GPU, but SNPs are processed in batches, and only a subset of a rank's SNPs reside on the GPU at any point (similar to out-of-core parallel programming, which has a long history [7], [8], [9], often with poor performance). This incurs many large transfers between host RAM and GPU RAM, increasing execution time.

The second approach minimizes such transfers by performing some computation on the CPU and transferring partial results to the GPU for completion. By carefully selecting computation for the CPU, we substantially reduce the size of the transfer from CPU to GPU. We focus on the second approach in this paper. Additionally, we investigate multiple approaches in which partial results are transferred.

One SNP per thread block.: Our baseline version of PLM is based on the second approach, where we split the execution of different operations between the CPU and GPU depending on suitability. For example, multiplying $SNP^T SNP$ and updating $X^T SNP$ in each iteration of PLM (see Section II) is of low arithmetic intensity, and is more efficient to execute on the CPU. Choosing to place such low intensity operations eliminates unnecessary and extra latency due to data transfer. Additionally, the resulting values of these chosen operations require substantially less storage than their arguments, which reduces the consumption of limited GPU memory.

Similarly, there are several other instances in the algorithm that can be more efficiently computed on the CPU. This is because some of the operations only need to be computed once for a given iteration of SNP

regressions, and some that need only be computed once for all kernel iterations. An example is in choosing the maximum F-value; the SNP corresponding to the maximum F-value is appended to the X matrix, which is then multiplied with its transpose. This process is only required to be completed once the max F-value is found among all GPU kernels processing all SNPs for a given regression iteration. By selectively performing this operation on the CPU rather than the GPU, we are able to reduce redundancy.

Two SNPs per thread block.: In an effort to increase the work per thread-block and reduce the memory consumption of the on-chip shared memory and overhead, we implemented a version of the PLM code in which threads work cooperatively to compute F-values for two SNPs rather than a single SNP per thread-block. The two SNP version maintains the same basic flow as discussed in the one SNP version; however, each computation performed is executed twice for two different SNPs.

The advantages with this strategy are (1) it reduces the shared memory requirements by a factor of two, because the same shared memory allocation used on the GPU for cooperative reductions among threads is used for both SNPs on a thread-block in this implementation, and (2) it reduces the number of required thread-blocks in half for a given number of SNPs. The disadvantage, however, is that it increases register utilization by about 50%.

Because the GPU is limited to having a maximum of 65,536 thread-blocks in a given grid dimension, increasing the number of thread-blocks essentially doubles our maximum processing capacity for each GPU kernel.

Single-precision.: We also explored the option of implementing a single-precision version of PLM. The accuracy of results as well as performance are important issues for scientific computing applications. Determining whether or not to use double-precision during the implementation phase can thus be a difficult choice to make.

Our analysis indicates that the single-precision implementation chooses exactly the same SNP at the end of each iteration to add to the design matrix, X. Furthermore, a comparison of the P values shows that the average error introduced is quite small (less than 0.0001), which means that we can improve performance (especially when parallelizing large data sets) by using single-precision for PLM, as discussed in Section IV-B.

IV. Performance Results

Our experiments were run on the Dell XD Visualization Cluster (named "Longhorn") at the Texas Advanced Computing Center (TACC). Longhorn has 256 nodes, each with 48GB of RAM, eight 2.5 GHz Intel Nehalem cores, and two NVIDIA Quadro FX5800 GPUs [10]. The FX5800 GPU is of compute capability 1.3 and contains 30 multiprocessors (MPs), each containing 8 streaming processing cores, 16,384 32-bit registers, 16KB of shared memory, 64KB (device) constant memory with a 16KB cache (per MP), and 4GB of global RAM. The source code was compiled with GCC version 4.4.1 and CUDA toolkit version 3.2.

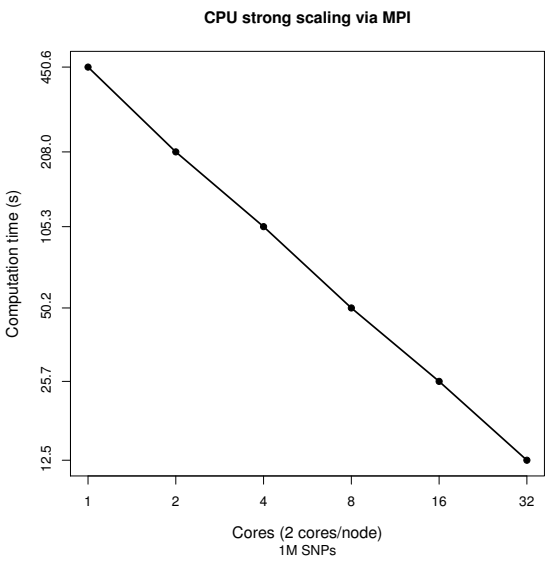

Figure 2. Performance at different node counts of MPI-only version of PLM.

We first present the results of the base MPI implementation of PLM. Following that, we present the results of various integrated MPI/GPU implementations of PLM on small data sets (100K SNPs of length 5k) that fit entirely within GPU memory. Finally, we present the results of integrated MPI/GPU PLM implementations on large data sets (1 million SNPs of length 5k) that do not fit within GPU memory. Here, it is necessary to investigate carefully several alternatives to the naive GPU parallelization of performing all computation on the GPU.

Our baseline MPI program (Figure 2) uses one core on each socket, even though the sockets have multiple cores. This is because we have not (yet) implemented an MPI/GPU PLM version in which we spawn one MPI process per core, where those MPI processes share the GPU. Such an implementation is more complicated because of a potential bottleneck at the GPU (or else a different programming model is needed). We leave this for future work; but for this paper, we used a one-to-one mapping of cores to

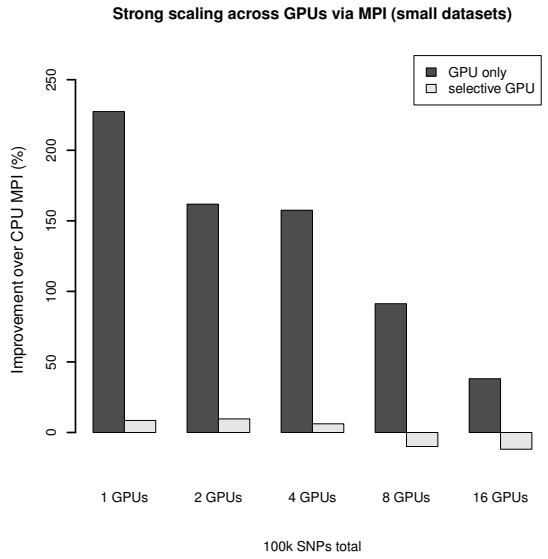

Figure 3. Performance of various implementations on small datasets (fewer than 100K SNPs per process each with length of 5k).

GPUs. Therefore, the baseline MPI program uses one core per socket.

As described in Section III-A, the forward regression algorithm with PLM lends itself to a straightforward MPI implementation. Figure 2 presents the results of MPI forward regression on CPUs (i.e., each MPI process is sequential and does not use the GPU). As expected, as forward regression is an "embarrassingly parallel" application from the point of view of distributed-memory computing, the performance scales linearly in the number of nodes. We refer to this as the baseline implementation when evaluating the GPU implementations.

A. MPI/GPU Implementation on Small Datasets

This section describes utilizing the GPU within each MPI process in the case that the SNP data assigned to each MPI process can fit entirely in GPU RAM. The results for a 100K double-precision SNP input of length 5k with relative improvement to the MPI-only version are presented in Figure 3. We observe that performing all computation on the GPU is best in this case (more than a two-fold improvement over the MPI CPU-only version). As forward regression with PLM is of low computational intensity, this improvement is due to the superior memory bandwidth of the GPU. In particular, the CPU implementation achieves 4.6 GB/s on a single core, and the GPU implementation for small datasets achieves 14.5 GB/s. The CPU and GPU in our experiments have theoretical maximum memory bandwidths of 25.6 GB/s and 102 GB/s, respectively.

The speedup of the version that selectively parallelizes PLM (see next section) on the GPU is much less for small data sets, showing an improvement of up to 7% in some cases. It also exhibits a slowdown as we scale up to 8 and 16 GPUs. As the number of GPUs increases, communication overhead increases; this overhead eventually causes a slowdown (beyond 8 nodes).

B. MPI/GPU Implementation on Large Datasets

This section describes utilizing the GPU within each MPI process in the case that the SNP data assigned to each MPI process is too large to fit within the GPU RAM. This presents significant problems with parallelizing the code within each MPI process. Table I illustrates the critical point: a naive GPU parallelization—in which all computation, but not all data, takes place on the GPU—performs not just poorly, but *worse* than the MPI CPU-only implementation. This is because execution time for this version is dominated by the time to copy the data from CPU RAM to GPU RAM over the PCI Express bus (this is the update of $X^T SNP$). The important lesson is that GPU parallelization must be done *selectively* for cases in which the working set does not fit in GPU RAM.

We evaluated four approaches by selectively parallelizing on the GPU only the matrix-vector multiplication, vector-vector multiplication, and reductions. We show the relative improvement of each approach, compared to the MPI-only version, for 1 million SNPs of length 5k, in Figure 4.

Double precision, 1 SNP per thread block.: When using 1 SNP per thread block and double precision, we achieved up to a 40% improvement over the MPI CPU-only version. It is important to note that the GPU-specific speedup was over a factor 3, but only a small portion of the code is being parallelized via the GPU.

Single precision, 1 SNP per thread block.: The best result achieved used single precision and 1 SNP per thread block. Here, we improved performance over the MPI CPU-only implementation by up to 64%, and 45% at minimum. Additionally, we found that the single-precision code performs at least 37% and up to 47% better than the double-precision code, with identical SNP selection at each iteration. This is because on the FX5800 GPU (compute capability 1.3), a multiprocessor contains eight processing elements capable of single precision arithmetic; however, it only contains one double-precision processing element [11].

Version/Number of Cores	1	2	4	8	16	32
MPI CPU-only	450.6	208.0	105.3	50.2	25.7	12.5
GPU-all	832.3	416.1	208.1	104.0	52.0	26.0

Table I
RESULTS OF GPU-ALL IMPLEMENTATION COMPARED TO MPI CPU-ONLY IMPLEMENTATION ON AT MOST TWO CORES PER NODE (TIMES IN SECONDS) FOR DIFFERENT NUMBER OF TOTAL CORES. THE GPU-ALL VERSION INCLUDES ONLY THE DATA TRANSFER TIME AS WELL AS THE TIME TO PERFORM THE MATRIX MULTIPLICATION AND SO IS A LOWER BOUND; STILL, IT IS *slower* THAN NOT UTILIZING THE GPU AT ALL.

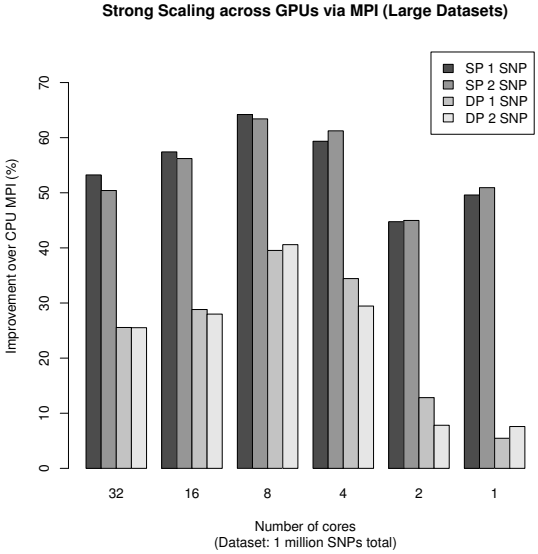

Figure 4. Performance of distinct MPI/GPU implementations on large datasets (1M SNPs, length 5k). SP indicates single precision and DP indicates double precision; 1 SNP and 2 SNP indicate whether a version uses 1 or 2 SNPs per thread block, respectively.

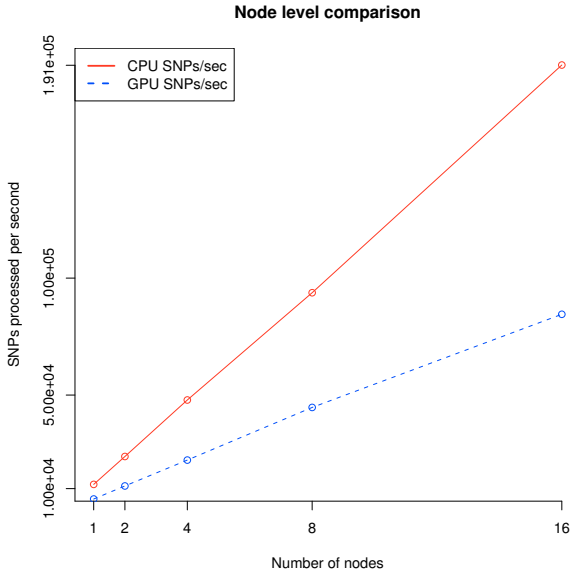

Figure 5. SNPs processed per second by using 8 cores/node with the MPI CPU-only implementation and 2 GPUs/node with the double-precision, 1 SNP per thread block implementation

Double precision, 2 SNPs per thread block.: We also experimented with using 2 SNPs per thread block. Here, we achieved up to a 41% improvement over the MPI-only version. Overall, there is no significant difference in terms of performance when using one-SNP versus two-SNPs per thread-block with double precision. (Again, this is because each multiprocessor contains only one double-precision processing element.) Increasing thread-workload and decreasing thread-blocks does not help in this strategy, because we are limited by the double precision capabilities of the device.

Single precision, 2 SNPs per thread block.: The single-precision, two-SNPs per thread block version also appears to have no significant change over the one-SNP code. Though there is additional work per thread-block, which translates to additional work per warp, the occupancy on a multiprocessor decreases due to the fact that each thread-block requires about 50% more registers for the two-SNP code. The two-SNP version would, however, be beneficial in small GPU systems where the number of available GPUs is limited. This is because there are kernel launches required with large data-sets that exceed the maximum allowable thread-block configuration for a CUDA grid.

C. Using all the available cores and GPUs on each node

The Longhorn cluster at TACC has 8 Intel Nehalem cores per node, and 2 NVIDIA Quadro FX5800 GPUs per node. Figure 5 shows the comparison of the MPI CPU-only implementation to the double-precision, one-SNP implementation when using all of the available cores on each node and all the available GPUs on each node.

D. Effect of varying SNP length

After the total number of SNPs, SNP length is the major driver of memory requirements. The forward regression algorithm is time- and space-linear in the

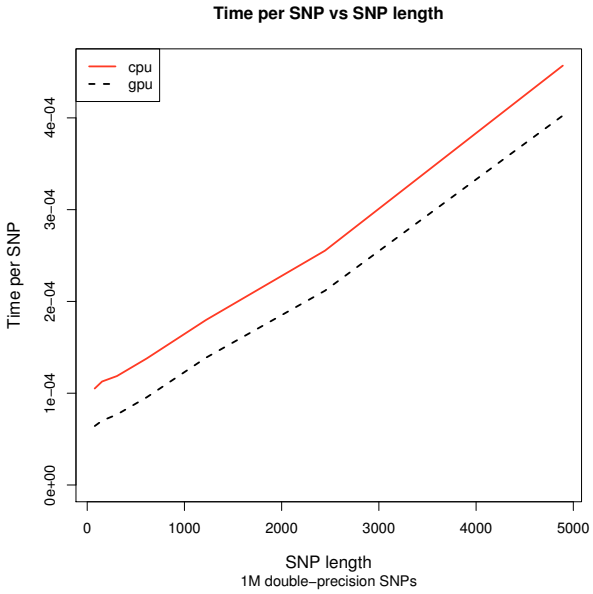

Figure 6. Effect of SNP length on computation time per SNP for CPU and GPU

number of SNPs, but also in the SNP length. However, a portion of the algorithm is independent of SNP length. This portion was selected for parallelization on the GPU due to its predictable memory requirements and FLOPS/word ratio. As a result, Figure 6 shows a consistent time difference between CPU and GPU implementations that is independent of SNP length. As SNP length increases, the portion of code that depends on SNP length begins to dominate execution time, and the advantage of the GPU diminishes.

V. Related Work

We are certainly not the first researchers to use GPUs to parallelize domain-specific applications. GPUs have been used to parallelize applications from linear algebra [12] to physics [13]. Closer to this work, GPU parallelization is common in systems biology [14] and sequence analysis [15], [16].

More specific to our project are GPU parallelizations of genome applications. For example, multifactor dimensionality reduction (MDR) is a technique that uses data mining to determine values of dependent variables from independent variables [17]. GPU parallelizations of MDR exist, including one applied to allow genome-wide testing for the disease ALS [18]. In addition, there was a test for bipolar disorder enabled by GPU parallelization of SNP-SNP interaction [19]. Davis et al. [20] studied GPU vs CPU parallelization for SNP ranking and concluded that with similar effort, CPU parallelization was competitive. In general, the approaches above do not specifically study GLM.

Finally, research groups have been studying porting the R statistics package [21] to GPUs [22], [23]. This would provide speedup to any program that used R. However, these packages are not fully developed (one is in beta, for example). Also, this approach is necessarily general, so it is likely that a hand-tuned approach such as ours will yield better performance. Considering the massive potential data sets we must consider, such a performance improvement may be worth the extra programming effort.

VI. Summary

This paper has described an integrated MPI/GPU implementation of PLM on both small and large data sets. While a straightforward implementation sufficed for small data sets, a more nuanced implementation was required for large data sets. For these implementations, we achieved speedups of 3.15 and 1.64, respectively.

Our future work will focus on handling ever-growing data sets. Specifically, the iPlant collaborative wishes to handle 50 million or more SNPs. This implies that the SNP data will not fit in memory, requiring a specialized out-of-core implementation. This will be challenging, adding complexity which may require re-implementation of the GPU kernels. In addition, we will work to improve the performance of the multiple-SNP GPU code.

References

[1] "iPG2P: Relating genotypes to phenotypes in complex environments," http://www.iplantcollaborative.org/grand-challenges/about-grand-challenges/current-challenges/ipg2p, 2010.

[2] D. Falconer and T. Mackay, *Introduction to Quantitative Genetics*, 4th ed. Prentice Hall, 1996.

[3] "iPlant project overview," http://www.iplantcollaborative.org/about/project-overview, 2010.

[4] "The iPlant Collaborative: Empowering a new plant biology," http://www.iplantcollaborative.org/, 2010.

[5] L.Wang, "General linear model for snp pair testing," https://pods.iplantcollaborative.org/wiki/display/~lwang/General+Linear+Model+for+SNP+Pair+Testing, 2012, [Online; accessed 17-December-2012].

[6] W. W. Hager, "Updating the inverse of a matrix," *SIAM Review*, vol. 31, no. 2, pp. 221–239, 1989.

[7] M. Kandemir, J. Ramanujam, and A. Choudhary, "Improving the performance of out-of-core computations," in *International Conference on Parallel Processing*, Aug. 1997.

[8] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koebel, and M. Paleczny, "A model and compilation strategy for out-of-core data parallel programs," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice or Parallel Programming*, Jul. 1995, pp. 1–10.

[9] T. C. Mowry, A. K. Demke, and O. Krieger, "A compiler-inserted I/O prefetching for out-of-core applications," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Oct. 1996, pp. 1–17.

[10] "Longhorn user guide," http://www.tacc.utexas.edu/user-services/user-guides/longhorn-user-guide, 2011.

[11] NVIDIA, "NVIDIA CUDA compute unified device architecture programming guide version 4.0," 2011.

[12] S. Lahabar, "Singular value decomposition on GPU using CUDA," in *International Parallel and Distributed Processing Symposium*, Apr. 2009.

[13] G. Shi, V. Kindratenko, I. Ufimtsev, and T. Martinez, "Direct self-consistent field computations on GPU clusters," in *International Parallel and Distributed Processing Symposium*, Apr. 2010.

[14] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units," *BMC Bioinformatics*, vol. 8, p. 474, 2007.

[15] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: Performance and design challenges," in *International Parallel and Distributed Processing Symposium*, Apr. 2009.

[16] H. Li and L. R. Petzold, "Efficient parallelization of stochastic simulation for chemically reacting systems on the graphics processing unit," *International Journal of High Performance Computing Applications*, vol. 24, no. 2, pp. 107–116, 2009.

[17] "Mulitfactor dimensionality reduction," http://www.multifactordimensionalityreduction.org/, 2010.

[18] C. Greene, N. Sinnott-Armstrong, D. Himmelstein, P. Park, J. Moore, and B. Harris, "Multifactor dimensionality reduction for graphics processing units enables genome-wide testing of epistasis in sporadic ALS," *Bioinformatics*, vol. 26, no. 5, pp. 694–695, Mar. 2010.

[19] X. Hu, Q. Liu, Z. Zhang, Z. Li, S. Wang, L. He, and Y. Shi, "SHEsisEpi, a GPU-enhanced genome-wide SNP-SNP interaction scanning algorithm, efficiently reveals the risk genetic epistasis in bipolar disorder," *Cell Research*, vol. 20, no. 7, pp. 894–897, Jul. 2010.

[20] N. A. Davis, A. Pandey, and B. McKinney, "Real-world comparison of CPU and GPU implementations of snprank: a network analysis tool for GWAS," *Bioinformatics*, Dec. 2010.

[21] "The R project for statistical computing," http://www.r-project.org/, 2010.

[22] "R+cuda: Enabling GPU computing in the r statistical environment," http://gpgpu.org/2009/06/14/r-gpgpu, 2009.

[23] "R+gpu," http://brainarray.mbni.med.umich.edu/brainarray/rgpgpu/, 2010.