Cost-effective GPU-Grid for Genome-wide Epistasis Calculations

B. Pütz; T. Kam-Thong; N. Karbalai; A. Altmann; B. Müller-Myhsok

MPI of Psychiatry, Statistical Genetics, Munich, Germany

Keywords
Epistasis, GPU, grid, computing

Summary

Background: Until recently, genotype studies were limited to the investigation of single SNP effects due to the computational burden incurred when studying pairwise interactions of SNPs. However, some genetic effects as simple as coloring (in plants and animals) cannot be ascribed to a single locus but only understood when epistasis is taken into account [1]. It is expected that such effects are also found in complex diseases where many genes contribute to the clinical outcome of affected individuals. Only recently have such problems become feasible computationally.

Objectives: The inherently parallel structure of the problem makes it a perfect candidate for massive parallelization on either grid or cloud architectures. Since we are also dealing with confidential patient data, we were not able to consider a cloud-based solution but had to find a way to process the data in-house and aimed to build a local GPU-based grid structure.

Methods: Sequential epistasis calculations were ported to GPU using CUDA at various levels. Parallelization on the CPU was compared to corresponding GPU counterparts with regards to performance and cost.

Results: A cost-effective solution was created by combining custom-built nodes equipped with relatively inexpensive consumer-level graphics cards with highly parallel GPUs in a local grid. The GPU method outperforms current cluster-based systems on a price/performance criterion, as a single GPU shows speed performance comparable up to 200 CPU cores.

Conclusion: The outlined approach will work for problems that easily lend themselves to massive parallelization. Code for various tasks has been made available and ongoing development of tools will further ease the transition from sequential to parallel algorithms.

1. Introduction

Genotyping is a well established method for studying genetic effects related to disease predisposition or manifestation as evidenced by the flurry of genome-wide analyses in the genetic journals these days. This is visible from the table of contents of such journals. To give an example, Nature Genetics in its December 2011 issue features 8 genome-wide analyses out of 17 total reports. High-throughput techniques allow the determination of currently up to approximately 5 million SNPs per individual [2, 3], which can be more than doubled by imputing unmeasured SNPs from reference data [4–6].

The effect estimation usually involves a linear regression of the genotype and the phenotype of interest. For current computers, this task is quite feasible, even on fully imputed genotype data (i.e., approx. \( n = 10^7 \) SNPs) for several thousand individuals. However, when moving on to pairwise interaction effects (epistasis [7]) this becomes an \( O(n^2) \) problem, and the number of interactions to be calculated rises to roughly \( 5 \times 10^{11} \). Only recent advances, especially employing parallel implementations, have made this problem tractable in reasonable amounts of time [8–13]. Higher order interactions are still out of reach due to the exponential increase in computational load.

In this paper, we will present our steps towards building a local high-performance GPU-grid and give a short overview over state-of-the-art approaches with links to relevant information. We hope that this overview will make GPU-based acceleration a less daunting venue for others with similar needs.

2. Implementation

2.1 Software

In order to evaluate the feasibility of porting any computationally intensive problem into a parallel computing environment, several considerations have to be made. In our case, this was assessed using standard tests on an 8 node CPU-based cluster. Convincingly, the theoretical maximum speed-up factor can indeed be reached consistently if one pays careful attention to the amount of workload that is assigned to each node and if latency related to memory transfer is handled properly. The next step was to evaluate whether the problem is more suitable for a multicore CPU or a GPU-based solution.

Many performance comparison reviews, often conflicting, are made between CPU and GPU based solutions. The fact that the performance gain is in general task
dependent was quickly realized. A CPU-based multicore solution is more attractive based on the level of maturity in the field of parallel computing and well-established evaluation protocols. As readily available CPU code can be accommodated with the help of application programming interface such as Open Multi-Processing (OpenMP) to run in a multicore environment, the learning curve involved is low. Evidently, to gain maximum efficiency, one would still need to optimize the script. This task, however, does neither have the added level of complexity related to learning a novel GPU-based programming paradigm nor the need to profile on host and device separately.

The individual processing tasks in the GPU, threads, are grouped into blocks. Each block has a dedicated amount of shared memory that can be used to share intermediate results among the threads. Once the jobs are distributed to the individual threads, it is important to retrieve and assign the results to their original problem once the parallel runs are completed. Each thread is assigned with an unique global ID that is created with an algebraic formulation based on the information created with the hierarchical organization of the threads by incorporating their block ID, block dimension and thread ID within each block. The overall unique thread ID is then used as reference for each unique problem that was treated in parallel. In subsequent instructions when updated data on the shared memory is required, it is necessary to synchronize the thread jobs to wait for the completion of all others before further computations should take place. In addition, writing and retrieving data from the global memory of the device is relatively slow compared to the throughput of the processing elements and hence is often a bottleneck in GPU implementations. In order to avoid this bottleneck, efficient implementations make use of the on-chip memory for reducing main memory traffic. In other words, it is best to create a scenario where the performance of the GPU kernel is computation-limited instead of bandwidth-limited. This facilitates the identification of the computation-limited routines in the program and new strategies to improve the performance can be implemented. Approaches to optimize GPU programming are similar to those undertaken in CPU programming. As the GPU works in a SIMD fashion (single instruction, multiple data, i.e., all threads execute the same instruction on their data), load balancing is evidently an important aspect to keep track of. Additionally, conditional statements in the code are best to be avoided as they may keep portions of the threads idle while others execute alternative branches of the code. It is also important to keep in mind that GPU kernel calls from the CPU are done asynchronously. Immediately after the kernel is called, CPU regains control and the GPU works on its own. This allows for computations on the GPU and CPU to take place separately. This ability to overlap the distinct computations on the host and device should also be exploited to its fullest in view of optimizing the overall program. The concept of porting the script to GPU has the risk of spending much time and effort in figuring out the complexities or phrasing and optimizing the problem suitable for GPU computing. This was more of an issue at the time when we embarked on this project. GPU computing was still very much in its infancy. Especially the debugging and monitoring tools were still rather rudimentary in comparison to their counterparts in the CPU regime. However, we did notice a high interest and curiosity among researchers in the field and have seen rapid progress in the development of software tools. Therefore, we expect to see a growing number of GPU-based solutions to become available [11–13] (see also [14]).

2.1.1 General Purpose Graphics Processing Unit (GPGPU) Programming

As the name implies, GPUs are primarily designed to accelerate graphics operations. In order to utilize them for other tasks, access to their hardware has to be made available to developers through some application programming interface (API). Currently, there are several approaches to allow programming for GPGPU computations.

2.1.1.1 Drop-in

The most direct access to GPU utilization is offered by modules or libraries that provide GPU-enabled functions that can directly replace existing functions. Examples are the Parallel Computation Toolbox for Matlab [15] or package gputools [16] for R [17] (see sec. GPU implementation for an example). On the level of compiler languages libraries such as CULA [18], an accelerated implementation of the well-known LAPACK and BLAS libraries, serve a similar purpose.

2.1.1.2 Hardware-specific Venues

CUDA (Compute Unified Device Architecture [19]) is the programming interface NVIDIA released in 2007 for its hardware. It can be seen as a C dialect with extensions for GPU control and data exchange between host and GPU memory. Code targeted for the GPU is developed in so-called kernels that are loaded into the GPU on demand to perform the computations. The recent release of NVIDIA’s Parallel Night debugger for Microsoft’s Visual Studio development suite has greatly improved the typically non-trivial development of these kernels.

Stream was released in 2008 for ATI GPUs and is now available from AMD as Accelerated Parallel Processing (APP) Software Development Kit (SDK) [20]. Since we have no compatible hardware, we have not explored this SDK.

2.1.1.3 Hardware-agnostic Approaches

OpenCL (Open Computing Language) has been developed as “open standard for parallel programming of heterogeneous systems” [21] and was released in December 2008. It can be thought of as a meta-level encapsulating CUDA, APP, and possibly others in the future that allows addressing all available resources, not only different GPU-architectures but also single- or multi-core CPU hardware, from within one portable application. Kernels can be compiled for different targets to provide access to those resources on machines where they are available. APP as well as CUDA are supported as GPU interfaces. We are currently exploring this option.

General-purpose compilers provide an interesting alternative, e.g., the (commer-
cial) PGI compilers. A very straightforward approach is offered by the Accelerator compilers for Fortran and C89 [22] that analyze specifically marked-up code sections (nested loops are likely candidates) and try to generate corresponding CUDA code without further intervention from the programmer. Thus, only minor changes to existing code may be required, another option that will have to be explored more thoroughly in the future.

Interestingly, the CUDA x86 compiler [23] offers the reverse path to run CUDA code on multi-processor CPU systems. It follows an approach similar to OpenCL, aiming to address both CPU – multiple CPUs with open multi-processing (OpenMP [24]) or the message passing interface (MPI [25, 26]) – and GPU resources (currently planned for CUDA only, with separate license).

OpenACC – The markup approach is also basis for the open accelerators (OpenACC [27]) initiative founded in November 2011. Its goal is to provide a standard similar to OpenMP for GPU programming.

2.1.1.4 SDK Availability

With the exception of APP for Mac OS all SDKs are available for Linux/UNIX, Mac OS, and Windows.

2.1.2 GPU Implementation of Epistasis Detection Algorithms

The R package gputools [28] had helped in overcoming the initial barrier for testing code on GPUs by providing drop-in replacements for standard functions that, transparent to the caller, internally make use of available GPU ressource. The gput Cor function (replacing cor for the calculation of correlation coefficients) had allowed a first glimpse of the possible speedup. By the same token, a novel statistical test was developed by phrasing the combinatoric problem as the difference of correlations of the two SNP pairs with the dichotomous phenotype [9]. The computation is well suited for GPU computing as identical operations are carried out for every SNP pair thus easily distributed over the GPU’s threads.

Limitations, such as GPU memory constraints, had to be worked around, but otherwise the implementation was straightforward as each SNP pair significance test can be conducted independently. This algorithm served as basis for a first implementation in CUDA [10]. The host system (CPU) is responsible for data preparation and parcellation of the task in chunks that the GPU can handle, sending these tasks to the GPU subsystem, collecting the results of the GPU computations, and their assembly into one result file. The GPU handles the computation-intensive part of calculating the correlations.

The latest implementation, GLIDE (GPU-based linear regression for detection of epistasis [29]), is the first linear regression method ported on the GPU to perform statistical measures on the bias (a bias term is required to remove any constant offset of the quantitative measure that does not embed any information which can be modeled by the independent/genetic variable(s)), univariate and interaction parameters of the basic epistasis model. The program allows for data in the continuous domain for both the phenotype (quantitative traits) and genotype (imputed SNPs).

2.2 Hardware

2.2.1 Nodes

As CUDA was supported by gputools at the time we started investigating GPU computing, we concentrated on respective graphic cards. NVIDIA offers two classes of cards, consumer level (GTX series) and professional (Tesla series). The latter offer advanced features addressing high performance computing (HPC) needs such as error checking on GPU memory and more memory on the cards, higher double precision float performance, unfortunately at a steep increase on price per performance. When comparing the two types, we found that the less expensive cards fulfilled all our needs and allowed us to build a much larger cluster than it would have been possible with Tesla cards. This choice will have to be made based on the end-user’s requirements.

We opted for custom-built compute nodes based on a high-performance PC motherboard (Asus P6T7WS Supercomputer Motherboard, 2.8 GHz Intel i7 930 with 4 cores/8 threads) that accepts 3 high-end GTX-level GPU cards for GPGPU computations (GTX580, offering 512 GPU cores) and a low-end card for the visual display on the monitor. With standard building blocks, this maximizes the amount of GPU that can be purchased for a given budget.

2.2.2 Grid

We currently have a cluster of four of these nodes. Hence, up to 12 GPU jobs can be executed simultaneously. A scheduling system, in our case Sun Grid Engine (SGE), and a collection of shell scripts was set up to ease the distribution of GPU jobs. To avoid problems addressing the GPUs on a single node such as a single GPU being simultaneously called by two separate jobs (collision), three separate queues were installed, each assigned to a specific GPU-slot across all managed machines.

The implementation of the above mentioned algorithms is based on one epistasis analysis per run, i.e., one phenotype vs. all possible SNP pairings. In order to enable further parallelization, the SNP lists can be split, naturally by chromosome. These tasks can then be submitted to the queuing system to be processed in parallel over the available GPUs.

3. Results

3.1 Performance

In order to compare the performance of a CPU vs. GPU approach, we benchmarked our GPU-based solution against the multicore CPU software tool FastEpistasis [8]. This tool is specially designed as an extension of the PLINK [30] epistasis module using either Symmetric Multiprocessing (SMP) or MPI as interface to distribute the work in order to take full advantage of any available CPU in a multi-core environment. In addition, FastEpistasis is not only programmed to take advantage of the multicore resource, it is also enhanced to run more efficiently than the standard PLINK computation.

We used simulated data to run comparisons of PLINK and FastEpistasis on our 8 CPU-core system to GLIDE on one GPU. As shown by Schüpbach [8], FastEpistasis’
speed scales linearly with the number of CPU-cores used. The noted performance can thus be scaled down to a single core by a factor of 8. The overall speedup factor noted between our proposed GPU solution and FastEpistasis on one CPU is in the order of approx. 200, for PLINK close to 2000. The factor in increased speed of the proposed GPU solution has shown to be robust with respect to the number of subjects in the study. As shown in Figure 1 and Table 1, a constant relative speed of PLINK/FastEpistasis to our GPU method is noted when evaluated for a range of 100–10000 subjects involved in the run. In other words, in this task the performance of a single GPU equals that of a cluster with roughly 200 CPU cores.

### 3.2 Financial Cost/Performance

One further aspect relevant when comparing the performance of the various implementations and architectures are the costs entailed. Using a snapshot of the current market prices of, e.g., the NVIDIA GTX 580 and the Intel i7 core 930 as comparison, for the computations involved in the epistasis search, the performance of the GTX580 is placed at approx. 4.50 GFLOPS/€ compared to 0.25 GFLOPS/€ for the i7. Thus, a cost benefit factor of approx. 20× in favor of the GPU is observed, where computing performance is measured in GFLOPS (floating point operations per second). The advantage of the proposed GPU system over a CPU cluster will become even more pronounced once the additional follow-up costs (e.g., power consumption, cooling and maintenance) are factored in.

### 4. Conclusion

With the availability of relatively inexpensive GPU-enabled hardware and recent advances in corresponding software packages, it has become possible to perform whole-genome epistasis analyses on a simple desktop workstation. In combination with some basic grid infrastructure this allows for easy scaling of the computing capabilities to need and budgetary constraints.
On the other hand, a considerable investment in software development may be necessary for problems where no solution is available. The speedup for other tasks mainly depends on how well the problem can be adapted to the targeted GPU-architecture. Hence, it might not be as high as observed in our case.

For our purposes, the installation of a custom-built GPU-grid proved to be the most cost-effective approach. The timeline in purchasing, setting up and optimizing an in-house GPU may vary in accordance to the problem at hand. However, using our experience as a guideline, the initial observation of the potential performance speed-up and implementation of the first tool were achieved within the first 1–2 months. The eventual tailoring and development of the tools are in fact, similar to many software programs, an ongoing process fueled by the numerous progresses made and libraries released within the GPU community as a whole.

One must point out, however, that it is rather uncertain what the eventual optimal solution would be as the competition between CPU and GPU continues. Manufacturers of CPUs have come up with attractive and competitive multicore solutions (e.g., the Many Integrated Core (MIC) Architecture from Intel recently released as Xeon Phi “Knights Corner”), while the GPU roadmap promises impressive speed boosts with upcoming new generations. The tendency for the moment is that a heterogeneous co-processing computing model between a host and a dedicated parallel computing device is the most likely fast and cost efficient solution.

Finally, due to the sheer number of features and possible tests involved, the classification of datasets in genetic studies has become an increasingly common problem to tackle from a computational perspective. Here, one needs to implement new search strategies to approach this problem. Some classification methods in the machine learning field such as Random Forest [31], Adaboost [32] and Support Vector Machines [33] seem inherently simple to parallelize, and can in turn be run on multiple cores to reduce the search time per feature combination involved. Implementation of such methods and development of other novel strategies, which can further speed up the search time involved, will be part of the project on the larger scale. These methods will lend themselves to genomics studies, but also to studies of gene-environment interactions as well as proteomic, transcriptomic and metabolomic studies, and for studies with massive amounts of phenotypes such as studies of the influence of biological features (such as genetic or metabolomic markers) on voxel-based whole-brain imaging data.

The source code for the implementations we developed for our GPU grid are available for download [9, 10, 29].

References
23. CUDA compiler [Internet]. Available from: www.pgroup.com/resources-cuda-x86.htm.