The Use of GPUs in Genomic Data Analysis

M.P. Coffey, R. Mrode and T. Krzyzelewski
SAC, Roslin Institute Building, Easter Bush, Midlothian, EH25 9RG UK

Abstract

Modern animal breeding datasets are large and getting larger, due in part to recent availability of high-density single nucleotide polymorphism arrays and cheap sequencing technology. High-performance computing methods for efficient data warehousing and analysis are under development. Storage requirements for genotypes are modest, although full-sequence data will require much more storage. Storage requirements for intermediate and results files for genetic evaluations are much greater, particularly when multiple runs must be stored for research and validation studies. Genomic evaluation using large datasets requires a lot of computing power, particularly when large fractions of the population are genotyped. Large datasets create challenges for the delivery of timely genetic evaluations which must be overcome in a way that does not disrupt service provision in the transition from conventional to genomic evaluations. Processing time is important, especially as real-time systems for on-farm decisions are developed. Modern graphics processing units (GPUs) found in consumer PCs offer animal breeding a means to compute genomic breeding values in reasonable time.

Keywords: Genomic selection, GPU, CUDA, matrix calculations

Introduction

Genomic evaluations are becoming relatively widespread due to the availability of low(er) cost genotyping and the concomitant increases in the number of animals genotyped. It is expected that this trend will continue leading to price pressure downwards on genotyping and a further increase in use. Improvements in imputation techniques further allows lower density (and cheaper) genotyping platforms to be used by individual farmers for genotyping cows consequently adding greatly to the number of animals and density of SNPs being analysed. These trends are likely to have at least 2 important outcomes that will affect those engaged in providing genetic evaluation services 1) the genomic datasets will rapidly increase in size and require specialised handling and computing algorithms and 2) farmers expectations on the turn round time for the provision of genomic breeding values (GEBVs) will rise. The consequences of these for the production of GEBVs are that computing demands will be substantial and rising and a reappraisal of computing strategies may be required to ensure the continued provision of timely and useful services. In the calculation of GEBVs in the UK, around 80% of the total computing time is expended on preparing (multiplying) and then inverting the G matrix for reliability estimation. This fact allows a concentrated effort on that particular point in the computing chain for seeking efficiency gains.

Problem

The biggest (current) computing problem to be solved for calculation of GEBVs is matrix inversion for matrices of size at least 20,000 x 54,000 and as imputation yields more data, inversion of matrices of 20,000 x 800,000. As more and more cows are genotyped the G matrix may exceed 30 or 40,000 within 1 or 2 years. In any case and for any genotype density, the problem is big for real time service provision.

Solution

There appears to be a relatively cheap and promising technology available to help animal breeders keep pace with compute demands of genomic evaluation. Advances in graphic processing units (GPUs) found in many consumer PCs driving the display has been high due to demands from computer games. This has led to a technology called CUDA (compute unified device architecture; NVIDIA 2011) that exposes the underlying GPU hardware to developers. The technology has

already been exploited by systems that have very high computing demand and especially for calculations that can be parallelised. Examples are in weather forecasting and fluid dynamics (Corrigan *et al.*, 2010). For animal breeding, matrix multiplication and inversion falls into this category and lends itself well to acceleration using GPUs.

Whereas CPUs are very versatile and can deal with many different devices and tasks, GPUs are extremely limited in their capability but extremely fast at matrix manipulation since they have many cores that can all compute independently. Thus in contrast to CPUs, they do not do much but do it very well. For example, in an NVIDIA Gtx 590 there are 2 GPUs with over 500 cores and 1.5GB RAM on each GPU. This provides over 1000 cores that can compute in parallel on one card. Multiple devices (GPUs) can be rack mounted and addressed in parallel by one or more CPUs thereby making many thousands of compute cores available to solve a particular computation.

Fortunately for animal breeders, existing legacy software can be used and adapted to exploit GPUs. A Fortran compiler is available from Portland Group (http://www.pgroup.com/) that enables automatic creation of CUDA kernels at compile time that run on the GPU. This is achieved simply by surrounding loops that have independent variables by a special pragma that is conveniently ignored by other compilers. The same compiler also compiles native CUDA Fortran code written by hand to exploit the GPU features. This development route yields greater speed improvements but is more costly in development time since it will involve re-engineering of existing code. C++ compilers are also available to achieve the same result.

The performance gains are not easy to achieve simply by small adaptations to existing code since the matrices are often too big for the amount of RAM available. A drawback of GPUs is that currently they have limited amounts of on-board RAM and CPU RAM is unavailable to them. A 7072 x 47280 matrix of real data type occupies about 1.24GB RAM. Any manipulation of the matrix requires 2 or 3 times that amount of RAM. In order to overcome the limitation of small amounts of RAM on GPU cards, a wrapper is needed that breaks down the problem into parts and solves each part separately on the GPU before passing results back to the CPU for assembly into the final answer. Suppose that $\mathbf{C} = \mathbf{AA}'$, where A is a 7,072-by-47,280 matrix of floating point numbers. The problem may be broken down into blocks for processing in parallel as:

$$\begin{bmatrix} \mathbf{A}_1 & \mathbf{B}_1 \\ \mathbf{C}_1 & \mathbf{D}_1 \end{bmatrix} \begin{bmatrix} \mathbf{A}_2 & \mathbf{B}_2 \\ \mathbf{C}_2 & \mathbf{D}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1\mathbf{A}_2 + \mathbf{B}_1\mathbf{C}_2 & \mathbf{A}_1\mathbf{B}_2 + \mathbf{B}_1\mathbf{D}_2 \\ \mathbf{C}_1\mathbf{A}_2 + \mathbf{D}_1\mathbf{C}_2 & \mathbf{C}_1\mathbf{B}_2 + \mathbf{D}_1\mathbf{D}_2 \end{bmatrix}$$

where $\mathbf{A}_1$, …, $\mathbf{D}_1$ are submatrices of $\mathbf{A}$, and $\mathbf{A}_2$, …, $\mathbf{D}_2$ are submatrices of $\mathbf{A}'$.

The inversion of large matrices also is a common problem in genetic and genomic evaluations, and research is underway at EGENES to develop a system in which a CPU-side process will determine GPU availability and then break-down matrices into suitably sized blocks for piecewise inversion. This should (in theory) allow for the inversion of any matrix in a way that will utilize all available computing resources, either locally or in a cluster setting. Such a routine would be portable across many hardware configurations and would exploit all available CPUs and GPUs.

The cost of breaking the problem down to exploit the GPU hardware must exceed the benefit from accelerated processing speeds. Passing blocks of data from the CPU to the GPU takes time and passing the processed results back takes additional time. Thus a form of intelligence is required in the wrapper to detect available GPUs and determine from the size of the computation task, the expected benefits to be gained from offloading to GPUs. This type of intelligence is present in many compilers that seek to alter the basic source code to allow efficient optimization of compiled code. Sometimes, and for some problems, it is simply too costly to break up the problem and pass it to the GPU and so processing continues as usual on the CPU.

Discussion

Genotype datasets are getting larger but we already have many tools for working with them at present. To ensure that this remains the case in the face of expected increases in

dataset size, prototype software must be designed to consider scalability at the outset, which is often not done with software used for research purposes. Some computational resources, such as memory, disk space, and processing cycles, are relatively inexpensive and so expenditure can solve the immediate problem. Programmer time is much more valuable, and speed should be measured as person-hours from problem to solution rather than simply as data processing time for a single component of the system. Programmer training need to look back 15 or 20 years when hardware was expensive and rediscover strategies that focus on coding finesse rather than raw computational speed. Good code is good code irrespective of computing power, and the last decade has seen the growth of profligate programming. Students often have never dealt with size or computing constraints, which is important when working with large datasets such as genotype data. Software engineering has evolved into a mature discipline, and we need to re-learn and apply good development practices that consider scalability at the outset. Animal breeders should seek out more formal training in programming, rather than depending primarily on self-learning. GPUs may be used to add computing power where it is needed.

Acknowledgments

NVIDIA are acknowledged for their generous gift of 6 CUDA enabled graphics cards for research.

References

NVIDIA Corporation. 2011. CUDA: Parallel Programming Made Easy. http://www.nvidia.com/object/cuda_home_new.html. Accessed August 9, 2011.

PORTING OF FEFLO TO GPUS

Corrigan A, Camelli F, L□ohner R, and Mut F. V European Conference on Computational Fluid Dynamics, ECCOMAS CFD 2010, Lisbon, Portugal,14-17 June 2010.

Table 1. Time to multiply a matrix by its transpose using a CPU or a GPU.

Rows x columns		**1x1**	**2x2**	**4x4**	**8x8**	**16x16**
CPU	**Matmul (seconds)**	**507s**	**512s**	**513s**	**534s**	**566s**
	Time increase		**101%**	**101%**	**105%**	**112%**
GPU	**mmul**	**x**	**x**	**70s**	**94s**	**143s**
	Time increase				**134%**	**204%**
GPU v CPU				**-86%**	**-82%**	**-75%**