Research Note

RN/12/08

Initial experiences of the Emerald: e-Infrastructure South GPU supercomputer

17 June 2012

W. B. Langdon

Abstract

The Emerald supercomputer contains more than a thousand CPU cores and several hundred nVidia Tesla. A genetic programming GeneChip datamining application which searches for non-linear gene based prediction of long term survival following breast cancer surgery was transferred without change and run on part of the Emerald cluster. At 7 giga GPopS^{-1} it is the fastest ever genetic programming application of this type.

Keywords: GPGPU, nVidia CUDA, Fermi Tesla M2090, parallel computing, LSF, unix, GP, Bioinformatics, data mining, gismo, Affymetrix HG-U133A and HG-U133B, GSE3494, evolutionary computation

1 Introduction

The Emerald super computer has recently been installed. It contains 372 top end nVidia Tesla GPU processors (see Figure 1). I will report performance of an existing genetic programming [Poli *et al.*, 2008] breast cancer data mining application on Emerald and various gotchas for the unwary novice user.

2 Background

Emerald is shared by the Science and Technology Facilities Council (STFC) and the e-Infrastructure South consortium (led by Anne Trefethen and comprised of Oxford, Bristol, Southampton Universities and UCL). The next section summarises the existing GPGPU data mining application. Then Section 4 shares some initial experiences of Emerald. This is followed by world beating performance results from it for two approaches to parallel running on Emerald (Sections 5.2 and 5.2). In Section 6 I discuss problems with file access and allocating GPU Tesla to jobs.

3 Genetic Programming and the Bioinformatics Data Mining Application

For three years (1987–1989) samples were taken from most of the women who underwent surgery for breast tumours in Uppsala. The biopsies were subsequently measured using Affymetrix GeneChips [Miller *et al.*, 2005] generating more than a million data points for 251 patients. We obtain these gene expression data via NCBI's GEO, checked them for spatial errors, quantile normalised them and then used genetic programming to datamine them eventually yielding a small predictive model [Langdon and Harrison, 2008]. (The normalised data are available via `http://groups.csail.mit.edu/EVO-DesignOpt/GPBenchmarks/uploads/Main/GSE3494/` .) The original genetic programming work used an nVidia GeForce 8800 GTX GPU (with 128 stream processors) and RapidMind C++ software. It was recently rewritten in CUDA and the original experiments re-run on a C2050 Tesla GPU donated by nVidia [Langdon,] (code available via FTP). The C2050 code has just been run without modification on Emerald.

The genetic programming approach [Langdon and Buxton, 2004] is somewhat unusual in that instead of trying to solve the datamining problem in one go it uses several phases in which multiple independent runs are used to select which of the gene expression variable convey enough information to be useful in evolving a final non-linear predictive model of breast cancer survival. (The models are quite small, on average they contain only 12.9 components.) The goal here was simply to demonstrate re-running this approach on Emerald. Other experiments might use different numbers of phases and different numbers of independent runs in each phase.

For the breast cancer data there are three phases (see Figure 2). The first two phases each consist of 100 independent runs, each with a population of 5 million non-linear breast cancer predictors evolved from generation zero to generation ten using the data from 91 women for fitness training to select good gene expression models. In each run all 5.2 billion fitness tests are done on the GPU (taking a total of about ten seconds). The other operations remain on the host (although these too might in future be run on the GPU). Calculating fitness (before GPUs) used to totally dominate run time. Now operations which used to be a trivial part of the total time are significant and often the host operations take four or five times as long as those on the GPU.

At the end of each run in the first two phases, 8 000 good non-linear models are harvested and the gene expression data they use is extracted. Only gene expression data from good models is passed to the next phase. There is a single run in the third phase. It takes the eight best of the original 1013888 variables filtered by the first and then second phases and generates the final predictive model.

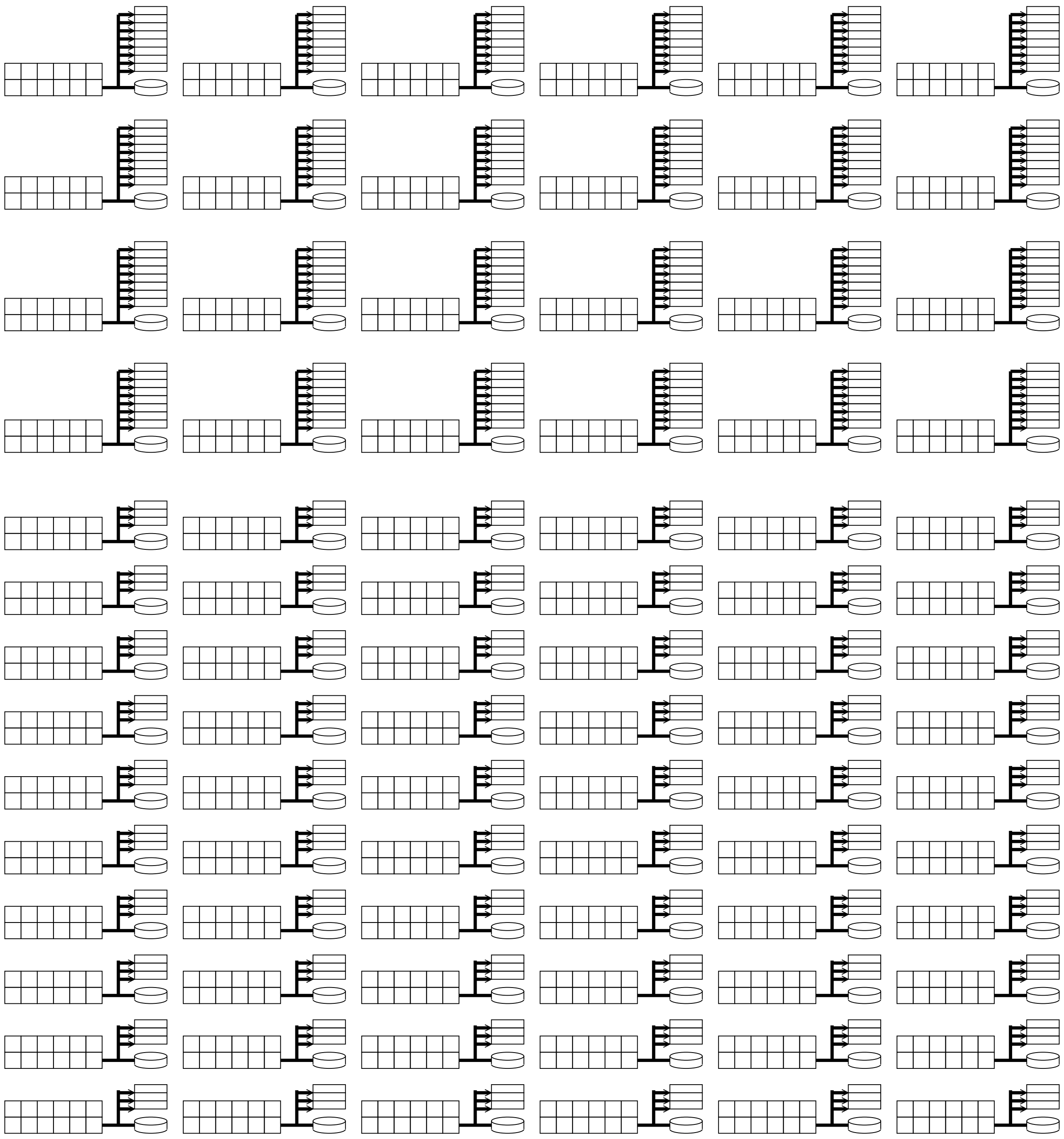

Figure 1: Schematic of Emerald GPU supercomputer. Each node consists of 6 twin-core CPU (squares), local disk and 3 or 8 nVidia Tesla GPUs. All 84 nodes are connected to a 135 000 GBytes Panasas storage system.

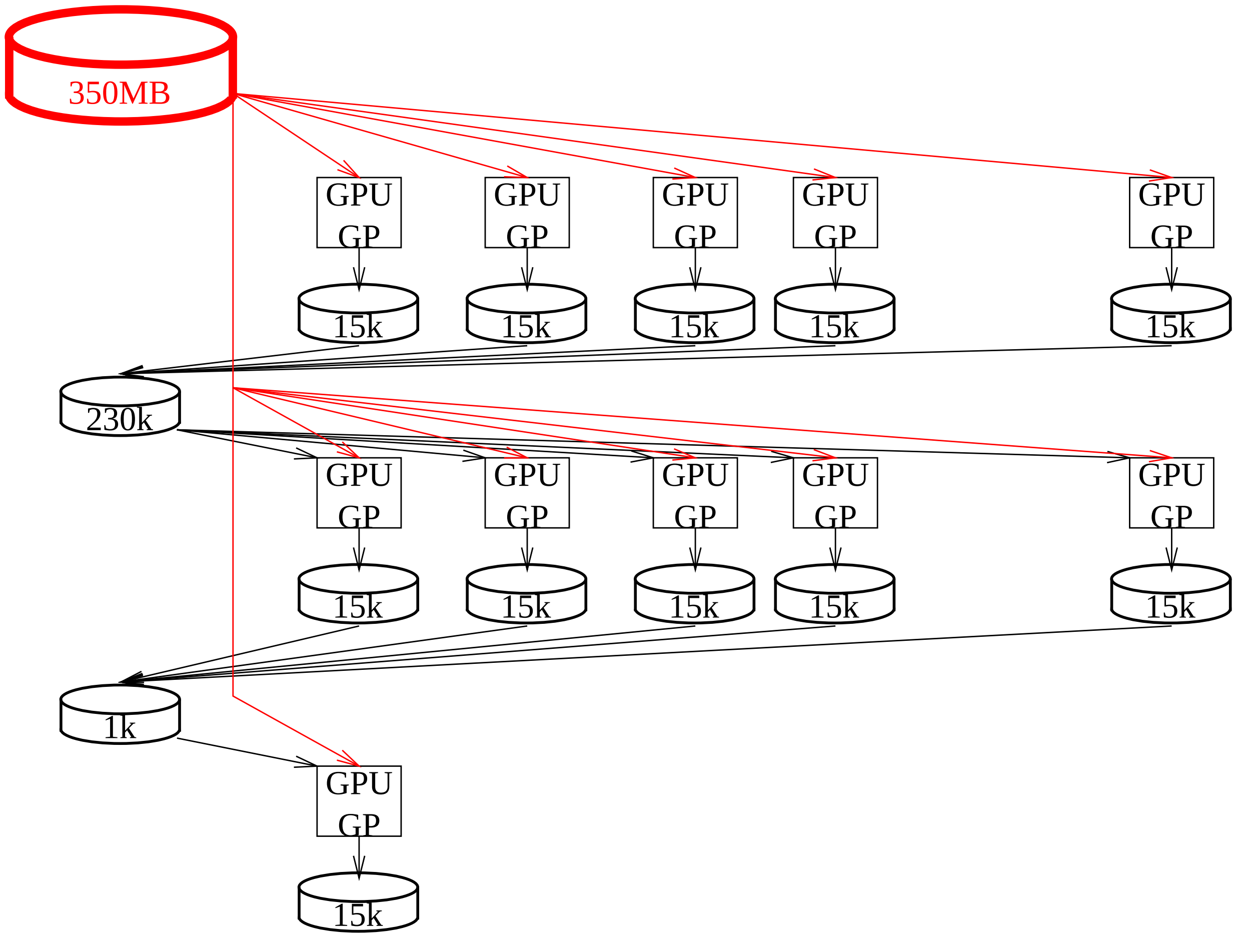

Figure 2: Schematic of major data flows when datamining breast cancer dataset. (Training data in red.) Only 5 of the 100 GPU GP runs in the first two phases are shown. The last phase (bottom) consists of a single GPU GP run.

4 Problems

4.1 Disk Server Bandwidth

The available bandwidth from the disk server to the GPU nodes seem to be low. When transferring the training data we observe a data rate of up to 4.3Mbytes/sec. Hopefully this is temporary problem since the installation has only just been completed.

It appears that the Emerald NSF disk system will happily cache even large files (like the genetic programming training data, 350 Megabytes) but does not broadcast data. So, if one hundred nodes try to read the same file, NSF simultaneously provides 100 copies of it. Which takes 100 times as long as to read it once. This means it can take more than half an hour for all jobs to read the training data.

4.2 LSF

The LSF batch queue software was written to share CPU cores. It does not seem to know about GPUs. Finding out which GPUs are in use is tricky.

4.3 Ganglia web pages

There are many Ganglia web pages which provide detailed information on the current and past loading of nodes in the Emerald cluster. However I found them hard to use and did not really get useful information on the GPUs from Ganglia.

4.4 Gotchas

There are a few little things which I found useful which are recorded in web page `http://www.cs.ucl.ac.uk/staff/W.Langdon/emerald/` for example:

- By default there is as usual no `noclobber` protection. If you are used to unix utilities not overwriting your files you may want to turn it on (`set noclobber`).
- The command to add nVidia tools to your unix path is simply `module load cuda`
- I have never found core dumps useful. Typically they are big files. Having large numbers of them created by multiple processes failing across Emerald for the same reason at the same time appears to really hit the file server bandwidth.

  The tcsh command to disable core dumps is `limit coredumpsize 0`

  (With the bash unix shell command line processor use `ulimit -c`).
- To make sure your script runs under unix tsch command line processor ensure the first line of the script is `#!/bin/tcsh`
- I found it useful to have a tcsh script for submitting arrays of jobs to LSF.

  The `-w` option on `bsub` is particularly useful for ensuring more than 100 jobs from a previous array of jobs have completed but `numdone` needs to know the job id of the previous array. E.g. with `-w "numdone($bestgenes,>=100)"` we have to supply a value for `$bestgenes`. I used an environment variable called `$bestgenes` and assign it the job id. This requires parsing the output of `bsub`. I wrote a script, `jobid.awk`, to do this but the essential part is:

```
bsub -J "bestgenes[1-$njobs]" bestgenes.bat > /tmp/bestgenes.tmp
if($status) exit $status;
setenv bestgenes `gawk '(split($2,t,"[<>]")){print t[2]}' \
                  /tmp/bestgenes.tmp`
if($status) exit $status;
```

(Notice the use of another global environment variable, `$njobs`, to control the number of jobs in the LSF array.)

- Although it is possible to read files held on the disk server directly into the GPU code, given the network delays (mentioned in Section 4.1 above) I found it useful to explicitly read files and save them on the local nodes `/tmp/` disk (using `rsync`) and then have the GPU code read from `/tmp/` Doubtless this imposes a small overhead, but separating file I/O and GPU processing makes it easier to find out why your super fast GPU code has snailed.
- I found tools like `bpeek` helpful.

5 Results

5.1 100 Jobs

The first approach was to rely almost entirely upon the LSF batch system to run GPU jobs in parallel and to synchronise them. There are two large job arrays, which are distributed about the Emerald cluster by LSF. Unfortunately some jobs find that the GPU that they had expected to use is not available and immediately fail. This imposes a low overhead but is messy and generally unsatisfactory.

A second LSF job is queued waiting for 100 of the first pass jobs to complete satisfactorily. The GPU jobs are pretty uniform. If running well they each take approximately the same time. Therefor synchronising all one hundred jobs should not lead to much processing resources being wasted. (Also LSF does not allocate exclusive access to 100 processors throughout the whole datamining exercise, it is free to pass idle processors to other Emerald users.)

The job to process the output of 100 first pass jobs is simple and does not consume much CPU time (and no GPU time). It condenses about 1.5MB down to 230 Kbytes which are read by the second array of jobs, which are all free to start as soon as it is done.

The second pass GPU jobs are very similar to first pass and (a part from reading the training data) take about the same time. They are followed by a second consolidation job (which is held waiting for 100 of them to complete) and then the final GPU job (which waits on the second consolidation job).

In many cases LSF reuses the same nodes. Therefore data copied in the first phase, is usually available with little overhead in the second and third phases.

Emerald has the compute power and GPUs to run each job in the three phases in parallel, which would give a total elapse time of the order of three minutes. Although we get considerable parallel running, due mostly to the time taken to transfer 91×1013888 floats, the last job terminated 47 mins 35 seconds (47:53) after the first job was started. This means on average Emerald interpreted 4 676 million genetic programming primitive operations per second. Certainly the fastest genetic programming performance on an essentially floating point problem yet published.

5.2 Three cn8g Jobs

The second approach was to accept that we are going to be data limited and instead of allowing the LSF job queueing system to allocated work freely we take more control. More-or-less the same structure was used, except instead of having two arrays of 100 jobs each, we have two arrays with 3 jobs in each and require those jobs to run on Emerald nodes with eight GPUs. Thus instead of the Emerald network having to distributed hundreds of copies of the training data we now explicitly copy it to just three nodes. (This took on average 4 minutes and 42 seconds (4:42), corresponding to $3.93\,10^6$ bytes/second.)

Instead of running just one GPU task, the three jobs have 34 or 32 to do. Internally the three jobs attempt to run eight GPU tasks in parallel and then use the unix `wait` command to synchronise between them. In most cases all eight run satisfactorily and the next eight can be started. Even if some of the eight GPU Tesla are not available the remaining GPU tasks take more or less the same time so synchronising them before starting the next GPU task is not too wasteful. Sometimes a GPU is in use and so the GPU task fails and it must be retried later. Although typically a job takes five cycles to complete, in one case it took 13. Obviously this delays the other jobs too. If this job had exclusive access to all eight GPUs then the problem would not have arisen and overall the second phase would have been more than thirteen minutes faster.

Instead of passing the output directly to the file server it is held on the local Emerald node's disk (in `/tmp`). This eases detection of GPU task failure and means we can do some preprocessing, so instead of transferring the whole output across Emerald's network only the summary data needed by the next phase is past to the file server. We still need to consolidate across all one hundred genetic programming runs, which is done by a single LSF job as before.

A typical first pass job running 34 GPU tasks now takes 10 minutes and 37 seconds (10:37) (about half of which is still spent waiting for the training data to arrive). However the second pass LSF job allocated an Emerald node with few GPUs free took 19 minutes and 37 seconds (19:37) giving a total elapse time of 31 minutes 48 seconds (31:48). This gives an average genetic programming (GP) speed of 7.14 billion GPop/s.

6 Discussion

In general purpose computing on GPUs (GPGPU) we are used to the idea that there is latency in transferring data within the GPU and between the GPU and its host PC, so it should not come as a surprise to find latency and bandwidth problems within Emerald. However the (current) bandwidth between the file store and the processing nodes is about a thousand times less than that within each node and the latency is measured in minutes rather than microseconds.

It is expected that shortly the connection between the file server and the Emerald network will be upgraded, improving bandwidth by a factor of ten. However the current speed ($4\,10^6$ bytes/second) seems to be well short of the theoretical speed of the existing connection (10^9 bits/second) suggesting there is a bottleneck elsewhere.

The computational and data pattern (see Figure 2) are specific to this problem but many applications will have a similar structure. They will have many operations that are able to proceed in parallel until specific points and data flows from the file store and a smaller volume goes back again. We are using an array of 91 by a million. This is not large in supercomputing circles. It is common for large dataset to be read only and have to be read at the same time. There are Internet broadcast protocols which avoid sending n-copies of the data. Is there something suitable for Emerald? Is each application going to need roll its own, perhaps with some processing nodes being dedicated to passing data onto the others?

Should we be looking to the GPU itself for inspiration. After all the GPU has the problem of large latency and restricted bandwidth. The GPU solution is mega-threading which assumes the memory system can cope with many independent requests for data and that the GPU can usefully process another thread whilst the first is blocked waiting for its data to arrive. Disks (and to some extent networks) are not like that.

Disks work efficiently with large (track sized) data requests. Emerald nodes (for this application) have more than enough RAM to buffer disk accesses so, unlike GPUs, we can avoid re-reading static data. This means we need only consider the first time the data are read. For us this means reading the training data, which has to be done by the time the GPU completes the first generation. This takes the GPU about 800mS (although in our case we also have some host start up activities which might also be overlapped with file I/O). In contrast, assuming the ten fold increase in network bandwidth mentioned above and some form of broadcast so the file is read only once, it will take about 9 seconds to transfer the training data. Thus we could be almost at the point where it might make sense to consider overlapping computation and file transfer for the first generation. Further improvements (perhaps file I/O bottle neck removal) might make overlapping file reads and computation attractive for other GPGPU applications on Emerald.

There are certainly things that can be done within our application to improve it. For example moving many of the current operations that modify the population which are done serially on the host onto the GPU to be done in parallel. However this involves significant development work and does not address the current file I/O bottleneck limiting the application's performance on Emerald.

The genetic programming datamining the GSE3494 breast cancer dataset represents about half an hours computation on an single M2090. I was worried this would be too small for Emerald. However I have been impressed by the responsiveness of Emerald and the LSF batch system in particular. It appears to react quickly to jobs finishing, is able to quickly start new ones in response and can cope with hundreds of simultaneous jobs.

7 Conclusions

An existing GPGPU Bioinformatics breast cancer datamining application has been ported to Emerald without code change. Despite Emerald only recently having been commissioned and in particular the Panasas disk subsystem not yet being up to full speed, across the whole process, the genetic programming interpreter averaged more than 7 billion genetic programming operations per second. This is the fastest floating point genetic programming application published.

The LSF batch system appears to work well at distributing jobs amongst Emerald's CPUs but it appears to be blind to GPU usage. Many of the LSF batch jobs described above fail immediately because their requests are rejected by the GPU they are trying to use. It would be good if there was some automated way (ideally as part of LSF) of allocating GPUs so that GPU tasks avoided trampling over each other.

With any new system there are pitfalls for the novice. Some of these have been described in Section 4. This application is not large by supercomputing standards but it is limited not by the speed of the GPUs or their host CPU but by reading data from the file server. Even after the planned upgrade in network connection to the Panasas file system, this is likely to be a generic problem and it would be nice to see a generic solution.

Acknowledgements

I am grateful for the assistance of Derek Ross, Mike Giles and Andrew Richards and njuffa (nVidia).

References

[Langdon,] W. B. Langdon. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In S. Tsutsui and P. Collet, editors, *Evolutionary Computation on Graphics Processing Units*, chapter 20. Springer.

[Langdon and Buxton, 2004] W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257.

[Langdon and Harrison, 2008] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing*, 12(12):1169–1183, October 2008.

[Miller *et al.*, 2005] Lance D. Miller, *et al.* An expression signature for p53 status in human breast cancer predicts mutation status, transcriptional effects, and patient survival. *Proceedings of the National Academy of Sciences*, 102(38):13550–5, Sep 20 2005.

[Poli *et al.*, 2008] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).