Proceedings of the 2014 Winter Simulation Conference
A. Tolk, S. Y. Diallo, I. O. Ryzhov, L. Yilmaz, S. Buckley, and J. A. Miller, eds.

MASSIVELY PARALLEL PROGRAMMING IN STATISTICAL OPTIMIZATION & SIMULATION

Russell Cheng

Mathematical Sciences
University of Southampton
Building 54, Highfield
Southampton, SO 17 1BJ, UNITED KINGDOM

ABSTRACT

General purpose graphics processing units (GPGPUs) suitable for general purpose programming have become sufficiently affordable in the last three years to be used in personal workstations. In this paper we assess the usefulness of such hardware in the statistical analysis of simulation input and output data. In particular we consider the fitting of complex parametric statistical metamodels to large data samples where optimization of a statistical function of the data is needed and investigate whether use of a GPGPU in such a problem would be worthwhile. We give an example, involving loss-given-default data obtained in a real credit risk study, where use of Nelder-Mead optimization can be efficiently implemented using parallel processing methods. Our results show that significant improvements in computational speed of well over an order of magnitude are possible. With increasing interest in "big data" samples the use of GPGPUs is therefore likely to become very important.

1 INTRODUCTION

The use of parallel computing in discrete event simulation (DES) work has been well discussed in the literature. For a review see Fujimoto (1990). However it has only been in the last three years that hardware based on graphics processing units (GPUs) has become sufficiently widely available and affordable to make personal desktop parallel computing possible. A good general introduction to the field in general is given by Kirk and Hwu (2013).

Currently there are a number of programming platforms and languages for personal parallel programming. These cover a range of approaches. At one end are platforms that require fairly detailed knowledge of the hardware architecture and which use a programming code, usually C-based, that sets out the programming details in explicit parallel fashion. A market leader is CUDA (Computer Unified Device Architecture), a massively parallel computer platform and programming language. At the other end are platforms which allow the user, coming from whatever field, to make use of their own original non-parallel code, but to include instructions in the code, typically pragmas, which enable the code to be compiled in parallel form using an appropriate compiler. Examples are *OpenML* and *OpenACC*. In use these act more like add-ons, to be applied in conjunction with a platform like CUDA, with the aim of saving the user from having to spend a large amount of effort in understanding hardware details and corresponding programming code. In between are platforms, like *Thrust* and *Arrayfire*, which are C-based, but are essentially large libraries of flexible constructs and subroutines which carry out commonly occurring calculations in parallel, such as adding together a large number of terms, a calculation that would have to be done serially when using just a single CPU. These platforms are not mutually exclusive, and can be used as part of a CUDA application.

CUDA has been widely used in partial differential equation applications arising mainly in the physical sciences, see Rumpf and Strzodka (2006). It has also been used in financial modeling applications (see for example Pagès and Wilbertz 2012) which deploy related differential models. CUDA has also been used in statistical methodology, see Matloff (2013, Chap. 14). To date its use in DES has been more limited.

At first sight it would seem that simulation experimentation is ideally situated for parallel processing. Consider for example the situation where we wish to make a large number of *independent* runs of a simulation model of the system under consideration and have a large number of CPUs, C say, available. If n runs are to be made each taking t units of real time to carry out then running them in blocks of C would enable the runs to be carried out in time nt/C compared with a required time of nt when the runs are made by just one CPU. There is the issue of variability in run length. There may be a variation in the real time needed to make different simulation runs, for example when simulating a prescribed number of customers processed by a system, but this variation is typically of order $\sqrt{t}$, which is much smaller than t when t is large. A good introduction to the statistical analysis of parallel simulations is given by Heidelberger (1986).

However a potentially more serious problem is the architecture of GPGPUs which places constraints on how the software has to be organized, if runs are to be made efficiently in parallel when statistically all the runs are mutually independent. We discuss this first, as at first sight it would seem that these constraints might severely limit the effectiveness of making parallel but independent runs of a DES simulation model.

We then, with the architecture of GPGPUs in mind, discuss two situations. In the first we consider simulation input or output analysis and show that there are situations where the calculations carried out in the analysis have a repetitive form that can be efficiently parallelized thereby significantly increasing processing speed, especially when large data sets are involved. We consider in particular the fitting of complex statistical models to a large data set using a method such as maximum likelihood. This is a numerical optimization problem requiring the maximization of the likelihood function, a quantity that is often required in statistical input or output analysis. Calculating the likelihood function can be time-consuming for large data samples, but the calculations set out in a parallel way when using a GPGPU. We show how this can be done and give a real practical example, where repeated calculation of the likelihood is needed, involving the fitting of a finite mixture model with normally distributed components to a large real data set obtained in a study of credit risk, showing that speed increases of well over 60-fold are possible when the sample size is large.

The second situation concerns the implementation of DES models using GPGPUs and whether this can give a worthwhile speed improvement. This is not an optimization problem in itself, however a DES model is often run repeatedly under different design variable settings in order to find the best operational set-up for the system. This in effect is an optimization problem and being able to examine the different variable settings simultaneously by making separate simulation runs in parallel is a simple and good solution to the problem.

In investigating the parallel execution of a DES model, we restrict ourselves to considering two simulation experiments using the same basic $M/M/c/k$ queue, where c is the number of servers available, and k is the maximum number of customers who can be held by the system, comprising both those being served and those waiting to be served. In one experiment a number of simulation runs are made of the model using a standard workstation CPU to carry out all the runs, so that the runs have to be made serially one after another. In the second experiment we use a GPGPU to make the runs. This allows the runs to be carried out in blocks in parallel. Our general discussion of how current GPGPUs operate would seem to suggest that making the runs in parallel in this way may not be entirely efficient. However our results for the $M/M/c/k$ queue show that speed increases of 30-fold are obtainable for this model.

Our main overall conclusion is that, though the hardware architecture would seem to handicap enhancing the performance of DES models themselves, in practice the gain in computational speed can nevertheless be impressive.

In the next section we outline some key hardware features of a GPGPU and how a programme written in the CUDA parallel is usually structured. In Section 3 we illustrate its use in a genuinely practical example involving fitting the previously mentioned finite mixture model, whilst in Section 4 simulation of the M/M/c/k queue is reported. This paper is an extension of the initial assessment of the use GPGPUs in simulation work given by Cheng (2014), but the examples provided here are different, thereby offering further evidence of the usefulness of GPGPUs in simulation work.

2 MASSIVELY PARALLEL PROGRAMMING WITH CUDA

We shall consider only the CUDA parallel computing platform. We have experimented with other approaches, especially *Thrust* and *Arrayfire*. However though these latter platforms are intended to save the user from having to take in, at least in too much detail, the specifics of a parallel programming platform such as CUDA, we found that, as far as statistical calculations and DES models are concerned, constructing a CUDA programming is relatively straightforward, and has the big advantage of enabling highly bespoke programs to be written.

We discuss only the hardware we actually used. This was a particular GPGPU in the Tesla range produced by the NVIDIA Corporation, one of the leading manufacturers in the field. Though Tesla GPGPUs have their origins in graphics cards, they are the first generation of NVIDIA cards designed with general purpose computing in mind. Though designed for general purpose programming, their architecture, called *Fermi*, is still strongly influenced by their graphics origin and this in turn determines to a large extent the form that the parallel programming code must take.

A very clear introduction to GPU programming with CUDA is provided by Matloff (2013, Chap. 5). We summarize some key points here.

A Tesla GPU is made up of an array of identical *streaming multiprocessors* (SMs). Each SM comprises a given number of individual *streaming processors* (SPs) usually called *cores* for short. In operation each core executes its own piece of code called a *thread*. Threads are arranged in *blocks* and the blocks are arranged in a *grid*.

The GPU is configured to operate with an accompanying CPU called the *host*, whilst the GPU is referred to as the *device.* The flow-controlling CUDA program is based in the host. The thread code is held in a routine called a *kernel*, which is held in the device. The usual form that an application takes when running just one kernel is given in Figure 1 showing that the host simply has to place input data into device memory then *launch* the kernel, i.e. set the kernel running, then wait for the kernel to complete execution after which control is passed back to the host.

An application can run several kernels. If kernels are called one after another the arrangement is as in Figure 1, except that after step D2 control returns to the host which launches the next kernel. Threads of the new kernel can use the output from the previous kernel as its input. This avoids unnecessary copying of data arrays, a very important point in view of the way memory is configured and accessed by the GPU, which we consider next.

The device memory has a quite complicated hierarchical form with six levels. The two most important are *shared memory* and *global memory*. Each SM has its own shared memory, which as the name implies, is shared by all threads running in that SM. This memory is on-chip so access is fast, but the memory size is small being typically only 16k bytes. This memory cannot be accessed by the host. Global memory is off-chip and shared by all threads of an application and accessible from the host; the contents are held for the entire application and across more than one kernel. However access is very slow involving hundreds of clock cycles per access. It is therefore very important that an application does not require threads to be constantly accessing global memory, as this will usually result in loss of all the speed gained by parallel running.

The other key feature to note is that threads in a given SM *do not run independently*. Blocks are assigned to specific SMs, with, in most applications, more than one block being assigned to each SM. Moreover the threads in a block are grouped into sets of 32, each called a *warp*. The key point to note is that each SM always runs the threads of a given warp together, with the threads having to execute instructions in what is called *lockstep*. This means that all the threads have to carry out the same instruction at the same time. The only variation allowed is that a thread must skip an instruction it cannot perform. Execution of warps is time-sliced with this actively controlled by their SM; if the threads of the currently active warp cannot be run, for example because they are waiting to receive data from global memory, then the SM will switch to another warp making active one whose threads can be executed. Warp *latency* is reduced if the block size is sufficiently large so that warps can be queued for execution. However if efficiency is to be maintained it is far more preferable that threads of a warp all execute the same code, and that this code does not contain any branching. This avoids what is termed *divergence* from occurring with different threads having to wait for long periods to do quite different things.

Host

H1 Calculate grid and block size to be used on the device.

H2 Allocate host memory for host input array.
H3 Allocate host memory for host output array
H4 Allocate device memory for device input array
H5 Allocate device memory for device output array

H6 Put input data into host input array
H7 Copy this data into device input array

H8 Launch kernel on the device using the calculated grid and block sizes.

Device

D1 Execute the threads in the device using device input array as input.
D2 Put output from threads into device output array
D3 Copy this to host output array
D4 Return control back to host

Figure 1: Typical form of a one kernel application.

This lock-step mode of operation would seem to prevent the possibility of different cores in the same SM from efficiently making its own run of a DES model. We have investigated this experimentally by making simultaneous but independent runs of a model of the M/M/c/k queue on separate threads. We have tried with a model capable of running in lock-step mode, and another where the model does not run in lock-step. The results, which are discussed more fully in Section 4, appear promising with good speed up using either model.

We consider also an aspect of DES work where lockstep parallelization will be highly effective. This occurs in the statistical analysis of input or output data used or generated in a simulation experiment. Consider a sample of input or output data $\mathbf{y} = \{y_1, y_2, ..., y_n\}$. Statistical examination of such a sample frequently involves subjecting each observation y_j to precisely the same set of calculations. Lock-step parallel implementation of the calculations is then not only feasible, but is quite easy to implement.

We shall give a detailed example of such an application in the next section. However before we do so we summarize and illustrate our discussion of this section by describing the particular Tesla GPU used in the numerical examples to be presented below.

The examples make use of the C2075 Tesla GPU. This has 14 SMs each with 32 cores making 448 cores in all. Each core runs at a frequency of 1.15 GHz, delivering a peak double precision floating point performance of 515 Gflops and a single precision performance of 1.03 Teraflops. As a warp always has 32 threads, having 32 cores per SM makes choice of blocksize rather straightforward. If we choose blocksize to be a multiple of 32 then each block divides neatly into a whole number of warps. In our experimentation we used blocksizes ranging from 64 to 512, not finding the precise value to be all that critical for good performance. The GPU has 6 Gigabytes of GDDR5 memory.

At this time of writing the C2075 card is less than 3 years old, but is already considered by NVIDIA to be ‘legacy’ hardware. It is certainly not the most powerful GPU available. However it is designed specifically for general purpose computing rather than just graphical rendering, and is sufficiently powerful to give some idea of the kind of benefits possible using massively parallel processing both in DES modelling and in statistical input/output analysis, especially for large data sets. We consider a statistical application in the next section.

3 A STATISTICAL ESTIMATION EXAMPLE

3.1 Calculating a Loglikelihood

As an example of our discussion in the previous section we consider the calculation of an important quantity used in many problems of statistical inference, namely the log-likelihood function

$$L(\boldsymbol{\theta},\mathbf{y})=\sum_{j=1}^{n}\log(f(y_j,\boldsymbol{\theta})) \tag{1}$$

of a random sample $\mathbf{y}=\{y_1,y_2,...,y_n\}$ of observations drawn from a continuous probability distribution with probability density function (PDF) $f(y,\boldsymbol{\theta})$ assuming this depends on a vector $\boldsymbol{\theta}=(\theta_1,\theta_2,...,\theta_p)$ of parameters. A good way to estimate $\boldsymbol{\theta}$ when it is unknown is the well-known method of maximum likelihood (ML). The estimator is that value of $\boldsymbol{\theta}$ which maximizes the likelihood function, that is

$$\hat{\boldsymbol{\theta}}=\arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta},\mathbf{y})\,. \tag{2}$$

In simple cases $\hat{\boldsymbol{\theta}}$ may be given by an explicit formula, however often the MLE is not available explicitly and has to be found numerically. A very convenient general numerical optimization method for doing this is the well-known simplex search procedure proposed by Nelder and Mead (1965) which calculates the loglikelihood for a sequence of parameter values $\boldsymbol{\theta}_0$, $\boldsymbol{\theta}_1$, $\boldsymbol{\theta}_2$,..., $\boldsymbol{\theta}_m$ that converges to $\hat{\boldsymbol{\theta}}$, the process stopping once $\boldsymbol{\theta}_m$ is judged sufficiently close to $\hat{\boldsymbol{\theta}}$.

The amount of computational effort needed depends on (i) the sample size *n*, (ii) the number of values of the loglikelihood that need calculating, and (iii) how complicated the expression for the loglikelihood is.

The dependence on the sample size *n* can be quite strong. We consider in detail a situation discussed by Cheng and Currie (2003) where a finite mixture model with *k* normally distributed components is fitted. This has PDF

$$f(y,\boldsymbol{\theta})=\sum_{i=1}^{k}w_i f_N(y\,|\,\mu_i,\sigma_i) \tag{3}$$

where $w_i>0,\ i=1,2,...,k$ with

$$\sum_{i=1}^{k} w_i = 1 \tag{4}$$

are the weights of the components whose individual densities are

$$f_N(y \mid \mu_i, \sigma_i) = \left(\frac{1}{2\pi\sigma_i^2} \right)^{1/2} \exp\left(-\frac{1}{2\sigma_i^2}(y - \mu_i)^2 \right). \tag{5}$$

Here $\mathbf{\theta} = (\mu_1, \sigma_1, w_1, \mu_2, \sigma_2, w_2, ..., \mu_k, \sigma_k, w_k)$ so there are $3k$ parameters. The individual terms $\log f(y_j, \mathbf{\theta})$ in the summation (1) have all to be recalculated for each $\mathbf{\theta}_m$, so there is a strong linear dependence on *n*.

3.2 Parallel Calculation of the Loglikelihood

We now consider how calculation of loglikelihood (1) can be done for the finite mixture model case (3) using GPGPU methods. Such parallel methods are increasingly used particularly by the high-energy nuclear physics community in fitting statistical models to data, see Jarp et al. (2011) for example. There are two main steps, both amenable to parallelization:

(i) Calculate the individual terms $l_j = \log f(y_j, \mathbf{\theta})$, $j = 1,2,..., n$ with $f(y_j, \mathbf{\theta})$ as defined in (3), (4), (5).

(ii) Calculate the sum $\sum_{j=1}^{n} l_j$.

The first step is very straightforward to do using parallel programming and can be carried out using just one kernel. This is because exactly the same calculations are used in evaluating each l_j, $j = 1,2,..., n$. The kernel therefore submits precisely *n* threads for processing by the device, with each thread calculating the value of precisely one l_j. As the form of the calculation is the same for every l_j this ensures that all active cores of a warp automatically work in lockstep. The thread code, given below (where $0.39894 \cong 1/\sqrt{2\pi}$, and tid = thread index *j*), is quite simple:

```
__global__ void LogLikelihoodKernel(double* y, double* mu, double* sig, double* w,
double* Res, int n, int k)
{
   // Each thread computes one term of the loglikelihood
   // adding it to the Res vector

   int i;
   double temp = 0;

   int tid = blockIdx.x * blockDim.x + threadIdx.x;
   if(tid >= n) return;

   for(i=1; i<=k; i++)
   {
      temp += w[i] * (0.39894/sig[i]) * expf(-(y[tid] - mu[i]) * (y[tid] - mu[i])
                       /(2.0 * sig[i] *sig[i]) );
   }

   Res[tid] = logf(temp);

   __syncthreads();
}
```

The integer *tid* is the index of the thread in the block whose value is being calculated. In our implementation we used a block size of between 64 and 512 threads. Note that this index runs from 0 to $n-1$ as is standard in C coding. However we have used indexing of i that runs from 1 to k to match the definition of the PDF given in (3). The functions expf() and logf() are fast versions of the exponential and logarithmic functions implemented in CUDA. The final instruction, `syncthreads()`, makes the kernel wait until all threads are finished before continuing.

The second step involves two or more kernels to evaluate a sum using a well known construction in parallel processing known as *reduction*. The basic idea is most easily explained when $n = 2^d$. The reduction is carried out recursively using $n/2$ threads initially. In step #1 each thread adds 2 terms yielding $n/2$ subtotals. In step #2 just half the threads, that is $n/4$ threads, are used each to add 2 of the $n/2$ subtotals. The process continues until step #d when just one thread adds the final 2 subtotals to give the required overall total. The process has therefore taken $\log_2 n = d$ steps. The case where n = 8 is illustrated in Figure 2.

The process is somewhat more complicated when n is not an integer power of 2. See Lensch and Strzodka (2008) for details. In our experiments we used a version based on the reduction routine `reduce6()` available as part of the CUDA software development kit, version 5.5, which is a highly optimized routine suitable for general n.

We have carried out a pilot experiment using a sample size of n = 32768 and calculated the loglikelihood function of the finite mixture model with normal components as given by equations (3), (4) and (5). We used a ten-component model, i.e. k = 10, and timed how long it took to calculate the loglikelihood 1,000 times using first the host and then the device. The host CPU, an Intel G2030 processor running at 3.00 GHz, took 11.19 seconds, whilst the Tesla C2075 took 0.094 seconds using a gridsize of 64 and blocksize of 512. The device was therefore 119 times faster. Given that the cores have a clockspeed of 1.15 GHz, it would appear that the speed improvement is about 70% of the maximum achievable. The likelihood is not usually calculated in isolation so the speed improvement just described represents an upper limit to what is achievable when calculation of loglikelihood values is only part of an overall statistical analysis carried out in a practical application. In the next section we discuss such an application, where calculations are essentially serial, but which require repeated calculation of the loglikelihood. Though there is a drop off in overall speed, repeated calculation of the loglikelihood forms the major part of the computational effort so that evaluating the loglikelihood using parallel routines is well worthwhile.

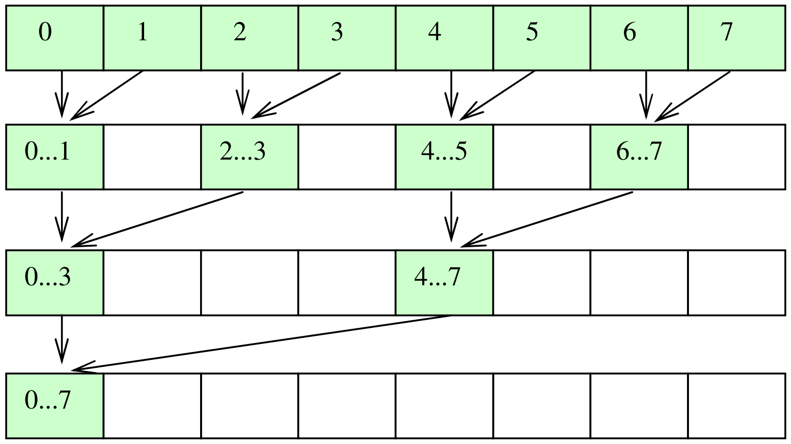

Figure 2: Reduction process for adding 8 numbers in $\log_2 8 = 3$ steps

3.3 Results

A fully working application has been developed using an Excel front-end interface for fitting the finite mixture model. Though not discussed here this working version takes a Bayesian approach fitting the k component normal mixtures for a selected range of k: $k = 1,2,...,k_{\max}$ where $k_{\max}$ is selectable. The Bayesian Information Criterion is also calculated for the maximized probability for each k. This allows the fits to be compared so that a best k can be selected.

We have used this application to fit the normal mixture model to an actual data sample comprising observations of 7051 standardized loss given defaults (LGDs) obtained in a study of credit risk. The frequency historam is not unimodal. A simple but meaningful assumption is that defaults come from a number of different distributions, and this is supported by the shape of the frequency histogram which is not unimodal. There is evidence of a small group of defaults whose values are all quite similar. A finite mixture model therefore seems to be an appropriate distribution to consider fitting. In our experiment we not only fitted the model to the full sample where n = 7051, but also to subsamples of size n = 500, 1000, 2000 and 4000 randomly selected from the full sample. We additionally created two large samples one which we call 2x, where each original observation is counted twice, and the other sample 4x, where each observation is counted four times. These two samples therefore contain 14102 and 28204 observations respectively. The application fitted the k-component normal mixture model for $k = 1,2,...,k_{\max} = 10$ to each of these 8 samples. For each k, the parameters were estimated numerically by ML estimation, using the host CPU, and using the Tesla C2075. For the sample comprising just the original observations, the best fit, as indicated by the lowest value of the Bayesian information criterion, occurred at k = 5. Figure 3 depicts this fitted 5-component normal model and the fitted densities of the five weighted components as well as the frequency histogram.

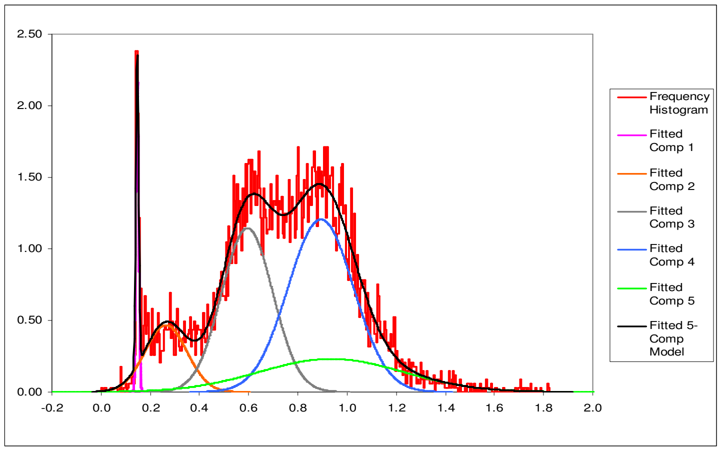

Figure 3: PDF of 5-component normal mixture model fitted to a credit risk sample of 7051 observations.

The times taken to carry out the estimation for all k values for each of the seven samples were recorded and these are shown in Table 1. It will be seen that using the GPU with parallel computation becomes progressively more effective. For the largest sample size of n = 28204, the CPU takes over 50 minutes to fit the parameters of all ten k-component models, whereas the GPU can do this in well under a minute.

Table 1: Total Computation Time in Seconds for fitting seven k-component mixture models, k=1,2,...,10.

n	500	1000	2000	4000	7051	14102	28204
CPU	19.1	50.8	99.9	359.0	547.6	1043	3435
GPU	7.46	13.2	12.3	21.6	21.6	27.4	50.2
CPU/GPU	**2.6**	**3.9**	**8.1**	**16.6**	**25.4**	**38.0**	**68.4**

4 M/M/C/K MODEL

We now consider use of the Tesla 2075 GPU in DES. We discuss only the M/M/c/k queue, where *c* is the number of servers and *k* is the maximum number of customers permitted in the system. It should be mentioned that in some special queueing situations, it is possible to arrange the calculations so that independent runs of the model can be calculated in lock-step. An example is discussed in Cheng (2014) in which the calculation of customer waiting times in the M/M/1 queue is calculated in lock-step. However for a general M/M/c/k queue, lock-step calculations for independent runs do not seem possible.

We do not give the complete kernel but just the main simulation loop included in the kernel. Though the coding of this loop has been carried out in a 'direct' fashion, it does deploy an events list albeit of a very elementary nature where the move to the next event needs only a test to see whether it is an arrival or a departure that takes place next. However even in this rudimentary form it does use an "if...else..." structure so that, run in parallel, the code, which includes random sampling, will not be executable in lock-step.

```
// ======================= Main simulation loop ======================
  while (0==0){
    if (timenextarr < timenextdep){  // === *** Arrival event *** ===
      nprevious = n;  // === n is the number in the system
      time = timenextarr;
      s = s + n * (time - tprev);  // Update area under "s" curve
      tprev = time;                // tprev = time of previous event
      if(n<k){ // === k is the max # customers allowed in system. Allow this arrival
        n++;
        if( n>c) numserversbusy=c; // === c is the maximum # of servers available
        else numserversbusy = n;
        a++;       // === a is the number of arrivals so far
      }
      if(a==Numcustmrsinrun){ // === Numcustmrsinrun is the # customers to be simmed
        timenextarr = huge_time; // == stops any more arrivals occurring
      }
      else{ // === schedule a next possible arrival time
        u = curand_uniform(&state[tid]);
        interarrtime= -Ta*log(u); // === Ta is the mean inter-arrival time
        timenextarr = time +interarrtime;
      }
      if(nprevious < k){
          u = curand_uniform(&state[tid]);
        servicetime= -Ts*log(u)/(double)numserversbusy; // === Ts = mean svce time
        timenextdep = time +servicetime; // === reset the next departure time
      }
```

```
        }
        else{                        // === *** Departure event *** ===
          time = timenextdep;
          s = s + n * (time - tprev);  // === Update area under "s" curve
          n--;                       // === decrease queuelength
          if( n>c) numserversbusy=c;
          else numserversbusy = n;
          tprev = time;
          numdeps++;                       // ===Increment number of departures
          if(numdeps==Numcustmrsinrun) goto endsimlabel; // === end the simulation
          if (n > 0){
            u = curand_uniform(&state[tid]);
            servicetime= -Ts*log(u)/(double)numserversbusy;
            timenextdep = time +servicetime;
          }
          else{ // === noone in queue, departure time not scheduled
            timenextdep = huge_time;
          }
        }
  endsimlabel: // === simulation run is over
        estimate[tid] = s/time;  //estimate of average queue length;
      } // === end of simulation loop
```

The code is self explanatory, except to note that `curand_uniform()` is a CUDA library pseudo uniform random variable generator with a period of over 2^{190} values. It is initialized by a call to `curand_init` which takes *seed*, *sequence*, *offset*, and *address of a state* as arguments. The *state* is initialized by the initialization call. The initialization sets the state to the state that would be arrived at after (2^{67}.*sequence* + *offset*) calls to `curand_uniform()` starting from the seed state. Thus use of the thread index *tid*, which is different for each thread, for the *sequence* value means that each thread will sample up to 2^{67} ($>1.475\times10^{20}$) values in its own segment of the total period before repeating values from another segment. Launching this kernel with arrival rate $\lambda = 4$, individual service rate $\mu = 1$, $c = 3$, $k = 5$, `Numcustmrsinrun` = 50000 `gridsize` = 14, `blocksize` = 512, produces 14*512 = 7168 runs of the queue completed in 2.06 seconds compared with 59.59 seconds for making the same number of runs of the same length carried out on the host CPU alone; a near 30-fold increase in speed when using the GPGPU. Both simulations estimated the mean queue length as 3.583 which is correct to 3 decimal places.

Though not reported here, other M/M/c/k queues with different arrival rates, service rates, *c* and *k* values were also simulated all yielding very similar improvements in speed. Thus running the code without lock-step still resulted in significant speed improvement in this simple DES example.

The example is obviously a very elementary example of a discrete event simulation, but the speed improvement observed encourages study of how effective GPGPUs would be in handling more complex systems. There is at least one area offering the potential of interesting research. This is the situation where each independent simulation run makes use of large data sets as input. The transfer of data from host to device memory is a bottleneck but this would be minimized if it is possible to partition the data into blocks with the simulation run length being divided into corresponding intervals in such a way that all processor cores are then able to operate on the same data block in the same interval. This would require the work of all processors to be efficiently synchronized so that they all complete processing of the same data block in the same corresponding interval. The simulation runs, though operating independently would nevertheless, all move forward together one time interval at a time with resynchronization carried out at the end of each subinterval.

5 SUMMARY

The examples gives a good indication of the usefulness of desktop massively parallel computing in the statistical analysis of simulation experiments and in DES. It shows that when sample sizes are relatively large then an order of magnitude or more of improvement is achievable in computing speed.

Our results are sufficiently encouraging for us to recommend further more extensive investigation. On the programming side, areas that need investigation are the best way to use the different forms of memory, especially in DES problems which require large data sets to be manipulated in the simulation run. In terms of areas of application, certain computer intensive methods of statistical analysis like resampling and bootstrapping are well suited to lock-step parallel calculations. Also analyses requiring the manipulation of large matrices would also be amenable to lock-step parallelization.

REFERENCES

Cheng, R. C. H. 2014. "The Use of Massively Parallel Processors in Simulation: An Assessment." In B.Tjahjono, C. Heavey, S. Onggo, and D-J. van der Zee, eds. *Proceedings of the Operational Research Society Simulation Workshop 2014 (SW14)*. To appear

Cheng, R. C. H., and M. S. C. Currie. 2003. "Prior and Candidate Models in the Bayesian Analysis of Finite Mixtures." In Chick S, Sanchez P J, Ferrin D and Morrice D J (eds). *Proceedings of the 2003 Winter Simulation Conference,* pp 392-398. IEEE, Piscataway,

Fujimoto, R. M. 1990. "Parallel Discrete Event Simulation." *Communications of the ACM*, 33(10): 30-53.

Heidelberger, P. 1986. "Statistical Analysis of Parallel Simulations." In Wilson J, Henriksen J, Roberts S (eds). *Proceedings of the 1986 Winter Simulation Conference,* pp 290-295. IEEE, Piscataway,

Jarp, S., A. Lazzaro, J. Leduc, A. Nowak, and F. Pantaleo. 2011. "Parallelization of Maximum Likelihood Fits with OpenMP and CUDA." *Journal of Physics: Conference Series*, 331, 032021

Kirk, D. B., and W. W-M Hwu. 2013. *Programming Massively Parallel Processors: A Hands-on Approach.* Elsevier: Waltham.

Lensch, H., and R. Strzodka. 2008. Massively Parallel Computing with CUDA: Lecture #6. http://www.mpi-inf.mpg.de/~strzodka/lectures/ParCo08/ accessed 2 February 2014.

Matloff, N. 2013. *Programming on Parallel Machines.* http://heather.cs.ucdavis.edu/~matloff/158/PL-N/ParProcBook.pdf accessed 2 February 2014.

Nelder, J., and R. Mead. 1965. "A Simplex Method for Function Minimization." *Computer Journal* **7**, 308-313.

Pagès, G. and B. Wilbertz. 2012. "GPGPUs in Computational Finance: Massive Parallel Computing for American Options." *Concurrency and Computation:Practice and Experience* 24(8): 837-848.

Rumpf, M., and R. Strzodka. 2006. "Graphics Processor Units: New Prospects for Parallel Computing." In*:* Bruaset A M and Tveito A (eds). *Numerical Solution of Partial Differential Equations. Chapter 3, Lecture Notes in Computational Science and Engineering,* v. 51. Springer: Berlin Heidelberg pp 89-132.

AUTHOR BIOGRAPHY

RUSSELL C. H. CHENG is Emeritus Professor of Operational Research at the University of Southampton. He has an M.A. and the Diploma in Mathematical Statistics from Cambridge University, England. He obtained his Ph.D. from Bath University. He is a former Chairman of the U.K. Simulation Society, a Fellow of the Royal Statistical Society and Fellow of the Institute of Mathematics and Its Applications. His research interests include: design and analysis of simulation experiments and parametric estimation methods. He was a Joint Editor of the *IMA Journal of Management Mathematics.* His email and web addresses are R.C.H.Cheng@soton.ac.uk and www.personal.soton.ac.uk/rchc