Technical Report CSTN-103

Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators

K.A. Hawick **A. Leist** **D.P. Playne** **M.J. Johnson**

Computer Science, Institute of Information and Mathematical Sciences,
Massey University – Albany, North Shore 102-904, Auckland, New Zealand.
Tel: +64 9 414 0800 Fax: +64 9 441 8181
Email: { k.a.hawick, a.leist, d.p.playne, m.j.johnson }@massey.ac.nz

Abstract

Generating quality random numbers is a performance-critical application for many scientific simulations. Modern processing acceleration techniques such as: graphical co-processing units(GPUs), multi-core conventional CPUs; special purpose multi-core CPUs; and parallel computing approaches such as multi-threading on shared memory or message passing on clusters, all offer ways to speed up random number generation (RNG). Providing fast generators that are also portable across hardware and software platforms is non-trivial however, particularly since many of the powerful devices available at present do not yet support full 64-bit operations upon which many good RNG algorithms rely. We report performance data for a range of common RNG algorithms on devices including: GPUs; CellBE; multicore CPUs; and hybrids, and discuss algorithmic and implementation issues.

Keywords: Monte-Carlo simulation; random number generation; seed management; configuration management; portability.

1 Introduction

Quality Monte-Carlo simulation studies rely heavily on reliable and high-performance random number generators. Many application codes are still hand-crafted for specific scientific problems, especially in areas like computational physics. These are often necessary for studying problems that require many machine cycles to attain the required statistical accuracy. These sorts of problem are embodied by simulation problems like that of the Ising model(Hawick et al. 2009) of a magnet (as shown in figure 1) that is used to study critical phenomena and where a bias or correlation pattern in the random numbers employed leads to the wrong answer.

For such applications it is often important that the code be portable to support taking advantage of any and all computer cycles that are available on a wide variety of hardware and operating system platforms. There are a number of practical issues, not widely discussed in the literature, that are concerned with fast, reliable and portable random number generator algorithm implementations. This paper presents some of these issues, particularly with regard to different processing acceleration devices.

Some relatively cheap accelerator devices such as Graphical Processing Units(GPUs), heterogeneous core processors such as the CellBE, or specialist processors such as Field Programmable Gate Arrays (FPGAs) or low power mobile devices like ARM, all offer potential price/performance advantages over conventional homogeneous core CPUs at the time of writing. Unfortunately many of these devices do not necessarily support full 64 bit operations, particularly for floating point algorithms. It is therefore interesting to consider what high-quality random number generator (RNG) algorithms can be deployed portably across a range of devices and what typical performance they can yield.

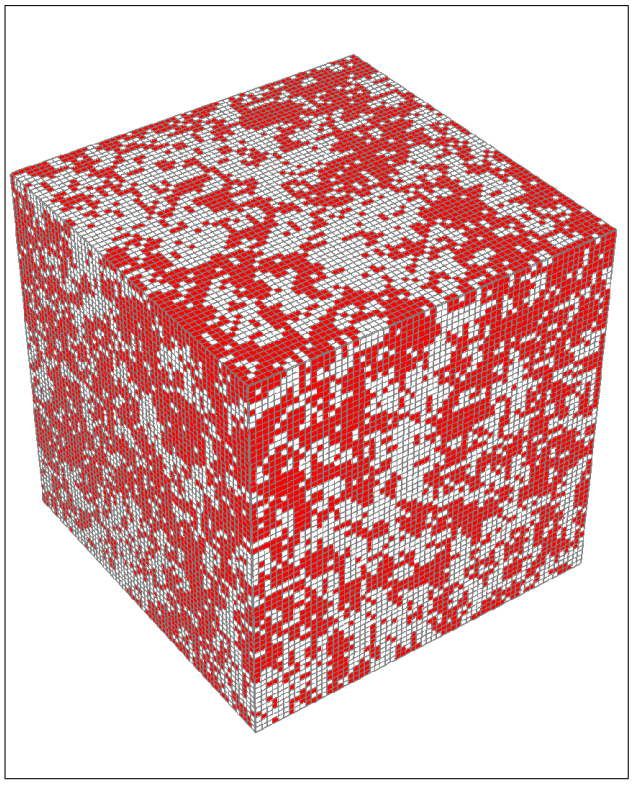

Figure 1: Monte Carlo generated Ising model on a $64 \times 64 \times 64$ cubic mesh at its critical temperature.

1.1 Historical Random Number Generation

Generating good quality fast random numbers(L'Ecuyer 2001, Brent 1997) on computers remains a long-standing challenge(Coddington 1994, Cuccaro et al. 1995). There is still an interesting algorithmic tradeoff space in which exist very high-quality generator algorithms such as the Mersenne-Twistor(Matsumoto & Nishimura 1998) that are significantly slower than those very-fast but lower-quality algorithms such as linear congruential generators. In between these extremes it is possible

to improve low-quality generator algorithms by adding lag tables and shuffles tables to further randomise or decorrelate the sequences of random deviates and indeed to combine several independent algorithms. Random number generators are usually formulated in terms of mathematical recurrence relations(Johnsonbaugh 2001) whereby repeated application of a transformation will project a number to another in an apparently random or decorrelated sequence - at least to the extend that any patterns discernible in the resulting sequence are on a scale that is irrelevant to the application using them.

There are some philosophically deep questions concerning what it really means for a sequence of deviates to be truly random. For most scientific purposes it is sufficient to say that they need to be sufficiently uncorrelated that when used for a Monte Carlo simulation or other application the deviate quality does not lead to an observable bias(Knuth 1997). Or put more simply – that the random number generator does not lead the applications programmer to the wrong answer. Various statistical tests, both at a straightforward level(Coddington & Ko 1998) such as the spacing test, scatter-plots, that detect obvious patterns or simple statistics are possible, as well as very specific application related tests that are highly sensitive to correlations.

A related issue is the period length of the generator algorithm. A few deviates generated to make a game program behaviour "interesting" to a player does not require a generator with a challengingly long repeat length. However, Monte Carlo calculations that may take weeks or months of supercomputer resources must have generators with very long period lengths. In the last 20-30 years of steadily increasing supercomputer performance, there has been continued interest in ever longer period generator algorithms. This often ties in with the need for more bits used in the generator. The 16-bit integer based generators of the late 1970s, were superceded by 24-bit (floating-point) algorithms such as the Marsaglia lagged-Fibonacci algorithm(Marsaglia et al. 1987), by the 64-bit integer based Mersenne-Twistor and in very recent times by 128-bit algorithms (Deng & Xu 2003) and even longer for cryptographically strong random number generation(Schneier 1996).

1.2 Generator Requirements

Randomness or lack of correlation amongst individual bits or patterns in the sequence of deviates is also very important. Some generators are known to have low correlation in some part of the generated bit patterns but not necessarily all, and therefore special operations can be used to only use those bit fields that are known to be decorrelated. Generally speaking if we have completely random bits we can generate random logical variables (obviously) but also integers and floating point to whatever precision we require. The reverse is not necessarily true and have a generator algorithm that produces a stream or sequence of integers or floating-point uniform deviates does not mean we can use them arbitrarily to reproduce random bits. A common target of many generator algorithms is to produce a sequence of random uniform floating point deviates – that is 32- or 64 bit floating point numbers on the range $[0.0, 1.0)$. A number of transformation algorithms(Hormann 1993) that can generate other statistically important distributions given a uniform stream of deviates are also well known. Some algorithms require particular low level data types to make them easily implementable. This can be an issue for some programming languages (such as Java) that may not offer access to low level data field features such as unsigned integers (Coddington et al. 1999).

In many scientific programs that use random numbers, repeatability is important at least at the testing phase of a program. Deterministic testing using a completely repeatable and reproducible sequence of deviates is desirable for debugging a simulation program. Quantum physical devices are now available(ID Quantique White Paper 2010) that can inject a highly random (but irreproducible) stream of deviates into a calculation with some excellent non-correlation behaviour. However this is not always desirable for reproducibility purposes, and at the time of writing there is still a significant overhead in obtaining deviates from such devices as they are typically implemented as I/O or bus-based devices and are not yet integrated onto processing chips.

This gives rise to another important criteria for random number generators - ideally they should be well engineered in terms of having plug-compatible software programming interfaces. This means that a code can be tested and implemented using any number of different generator algorithms with little code change required. A further complication is that for many modern programs the random number generation must be part of a parallel computation(Coddington & Newall 2004, Newell 2003). This brings its own special problems concerned with ensuring independent processors have independent decorrelated streams of deviates, and it can make complete deterministic reproducibility impossible to guarantee without some sort of parallel synchronisation to avoid timing drifts between parts of a parallel computation. Some generator algorithms are more amenable to parallelisation than others depending upon the memory structure of the lag-table or whether the algorithm supports long sequence jumps that would allow separate processors to be initialised far apart in a shared (long) sequence.

In summary then, the field of computer generated random number algorithms is one of "horses for courses" – there is no single best algorithm that will satisfy all requirements. It is therefore of worth to review some algorithms in common use and their implementation on parallel computational systems and devices.

1.3 Paper Outline

In our present paper we give algorithmic details for implementing several different generator algorithms on different devices with various parallel programming models. In section 3 we illustrate some key algorithm implementations in CUDA and in other parallel frameworks including conventional multi core CPUs with both POSIX and Intel threading; single GPUs using a data-parallel strategy; multiple GPUs in a cluster; and CellBE processors. We also discuss how the algorithms were timed on the various platforms. In Sections 4 and 5 we present some detailed timings for various generators and discuss the implications in Section 6. Finally in Section 7 we offer some conclusions on good algorithmic choices for computational science applications and directions for future development of random number generators, given the current trends in parallel compute devices.

2 Accelerators Architectures

Rather than producing faster machines by simply making the Central Processing Unit faster and more powerful, architectures are being developed with more specialised accelerators that can perform some specific computation much faster. We describe the architecture for two accelerators, Graphical Processing Units and Cell Broadband Engines.

Graphical Processing Units (GPUs) are proving to be very powerful processing accelerators for many scientific simulations and calculations and therefore in this paper we implement and test random number generators on GPUs using data-parallel techniques. We employ NVIDIA's compute unified device architecture (CUDA) programming language. CUDA supports very efficient code that fits the hardware, although the ideas and principles extend to the Open Compute language (OpenCL) specification(Khronos Group 2008) which is more widely supported by different vendors and devices. Some work has been done already on some generators for CUDA and GPUs(Langdon 2009, Giles 2009) and for other multicore parallel processors such as the STI Cell Broadband Engine (CellBE) (Bader et al. 2008) but with less emphasis on the topical issues of portability, performance tradeoffs and lack of 64-bit support.

2.1 Graphical Processing Units

Graphical Processing Units or GPUs have emerged in recent years as a very popular accelerator. This can be attributed to their reasonable prices, high computational throughput, common availability and relative ease of programming. Driven by the demands of modern 3D game graphics, both NVIDIA and ATI have developed highly parallel architectures to provide the processing required to supply these graphics in real-time. This architecture can be seen in Figure 2.

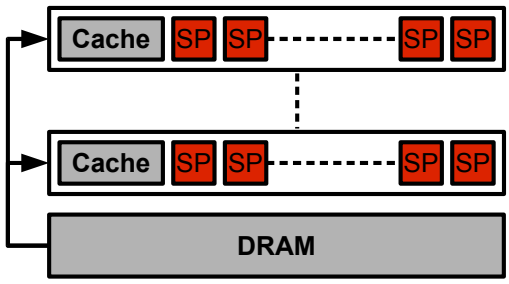

Figure 2: GPU architecture

GPUs contain many scalar processors (SPs) organised into multiprocessors (MPs). In the modern Fermi-based GeForce 400 series cards, each MP contains 32 SPs whereas in previous generations each MP contained only 8 SPs. GPU hardware can manage many thousands of threads as well as schedule and execute them on these multiprocessors. The multiprocessors can execute instructions independently from each other but the scalar processors within must execute the same instruction at the same time, this model is known as single instruction multiple thread (SIMT).

The main performance consideration for this architecture is how memory is accessed. All multiprocessors can access the main global memory (DRAM) of the GPU, however they also have some fast on-chip memory that the SPs within that multiprocessor can access. This allows SPs to reduce access to global memory and share information. Correct use of these on-board memory types has generally had the most impact on performance and has often been the main challenge of programming GPUs.

However, the release of Fermi-based GPUs has loosened the restrictions of global memory access. These devices now have an automatic cache structure similar to that seen on most CPUs. This cache structure makes it easier to achieve high performance on such devices, while still giving the developer the option to fine-tune his code to explicitly use the fast on-chip cache where necessary.

It is often desirable to use multiple GPUs to achieve a higher computational throughput. Simply using multiple GPUs is relatively easy as each GPU connects to a host thread, however if they must communicate or share information it can become more of a programming challenge. GPUs cannot communicate directly and all information sent between them must go through the host CPU. This involves copying the information from the GPU to the host, exchanging it with the other host thread and then copying it to the other GPU.

2.2 Cell Broadband Engine

The Cell Broadband Engine (CellBE) has been less successful as an accelerator than the GPU but still represents an interesting target in terms of accelerator development. This architecture has one traditional processor (the PPE) with 8 less powerful cores (SPEs) that it can delegate tasks too. These cores are all connected to the main memory of the cell and exchange messages through the Element Interconnect Bus. This architecture is shown in Figure: 3.

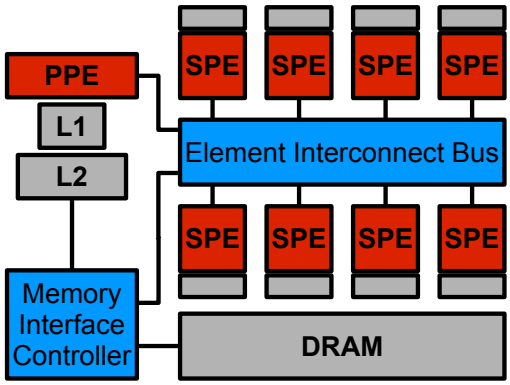

Figure 3: Cell Broadband Engine architecture

The major disadvantage encountered with the CellBE architecture was the programming API. CellBE applications require the programmer to manage the distribution of tasks and the exchange of data explicitly. Also to make full use of the CellBE processing power, the problem must be reworked to allow the SPEs to perform the computation in vector form. In many applications this can present a significant programming effort to rearrange the calculation to a vector form

3 Random Number Generators

Random number generation is one of the most widely used facilities in computer simulations. A number of different algorithms are widely used(L'Ecuyer 2001, Marsaglia 1984), ranging from fast but low quality system supplied generators such as the rand()/random() generators available on Unix(BSD 1993) systems to slower but high quality 64-bit algorithms such as the Mersenne Twistor generator(Matsumoto & Nishimura 1998). Marsaglia's lagged-Fibonacci generator(Marsaglia et al. 1987) is a 24-bit algorithm that produces good quality uniform deviates and which has been widely used in Monte Carlo work(Binder & Heermann 1997). It is convenient for our purposes in this present paper as not all our target accelerator hardware platforms uniformly support 64-bit floating point calculations.

3.1 CPU - Sequential Algorithm

The Marsaglia lagged-Fibonacci random number generator (RNG) has been described in full elsewhere(Marsaglia et al. 1987), but in summary the details are given in Algorithm 1, which we provide for completeness.

Algorithm 1 Marsaglia Uniform Random Number Generator, where an initialisation procedure sets the values as given below, and fills the lag table with deviates. The *id* and **signal** are not required for the sequential algorithm, but are used by the pThreads implementation described below.

```
function generate(id, seed)
  declare u[97]
  declare i ← 96
  declare j ← 32
  declare c ← 362436.0/16777216.0
  declare d ← 7654321.0/16777216.0
  declare m ← 16777213.0/16777216.0
  initialise(u, seed)
  for n ← 1 to N do
    uniform(i, j, c, d, m, u)
  end for
  signal complete(id)
```

Algorithm 2 Marsaglia Uniform Random Number Generator, each call will generate a single random number.

```
function uniform(i, j, c, d, m, u)
  declare result ← u[i] − u[j]
  if result < 0 then
    result ← result + 1
  end if
  u[i] ← result
  i ← i − 1
  if i < 0 then
    i ← 96
  end if
  j ← j − 1
  if j < 0 then
    j ← 96
  end if
  c ← c − d
  if c < 0 then
    c ← c + m
  end if
  result ← result − c
  if result < 0 then
    result ← result + 1
  end if
  return result
end function
```

Where i, j index a lag table which is shown here of 97 deviates, but which can be any suitable prime, subject to available memory and where c, d, m are suitable values.

A number of optimisations for this sort of random number generation algorithm are possible on the various implementation platforms. One obvious one is to synchronise a separate thread that can produce an independent stream of random deviates that are consumed by the main application thread. Other algorithms, whose descriptions are beyond the space limitations of our present paper, generate whole vectors or arrays of deviates together using a SIMD approach which can be used in applications that have similarly shaped work arrays or objects such as images or model data fields.

3.2 Multi-Core: POSIX Threads

The pThreads implementation of the lagged-Fibonacci generator launches multiple threads that each generate separate streams of random numbers. To do this each thread creates and initialises its own lag-table with a unique seed. The threads can then simply generate random numbers using this unique stream and the same `uniform` function as described in Algorithm 1.

Each thread that is created will generate `N` random numbers and then signal the main thread that it has completed its work. This code merely generates random numbers and does not make any use of them but it is assumed that any pThreads application that uses random numbers would make use of them within this thread loop.

3.3 Multi-Core: Threading Building Blocks

Like the pThreads implementation, the TBB implementation of the lagged-Fibonacci generator creates a number of independent RNG instances to generate streams of random numbers. However, the RNG instances are not associated with a particular hardware thread. Instead, they are each contained in a structure that can also store additional, application specific information related to the RNG instance. For example, it may also contain a pointer to an array that temporarily stores the generated deviates for later use, along with the array length. The structures are pushed into a vector after their RNG instances have been initialised. See Algorithm 3 for a description of this initialisation process.

Algorithm 3 Initialising the TBB implementation of Marsaglia's random number generator. The parameters to the function are the seed s_0 and the desired number of RNG tasks t.

```
function initialise-tbb(s_0, t)
  declare V //vector
  declare r_0 ← new RngStruct
  initialise(r_0, s_0)
  for i ← 1 to t do
    declare r_i ← new RngStruct
    declare s_i ← uniform(r_0) * INT_MAX //seed
    initialise(r_i, s_i)
    append r_i at the end of vector V
  end for
  return V
```

The parallel random number generation using these RNGs is invoked by passing the begin and end iterators of the vector to TBB's `parallel_for_each` function, together with a pointer to a function that takes the structure type as its only argument. TBB applies the given function to the results of dereferencing every iterator in the range [begin,end). This is the parallel variant of `std:for_each`.

The function called by `parallel_for_each` can then use the RNG instance passed to it to fill the array or array range specified in the same structure or to immediately use the random numbers in the application specific context. The process remains repeatable even though the thread that executes the function with a particular RNG structure instance as parameter can be different every time `parallel_for_each` is called.

TBB's task scheduler decides how many hardware threads are used and how they are mapped to the given tasks. While a larger number of RNG instances allows the code to scale to more processor cores, it also increases the overhead introduced by switching

tasks. If there are no other processes or threads consuming a significant amount of processing resources, then setting the number of RNG instances equal to the number of hardware threads gives the highest and most consistent performance in our tests. If, however, other threads are using some processing power, too, then splitting the problem into a larger number of smaller tasks gives the task scheduler more flexibility to best utilise the remaining resources.

3.4 GPU - CUDA

The CUDA implementation of the lagged-Fibonacci random number generator is based on generating a separate stream of random numbers with every CUDA thread. This approach, referred to as CUDA 1, is repeatable and fast as race conditions are avoided and no communication between threads is required. Algorithms 4 and 5 illustrate the implementation of Marsaglia's algorithm in CUDA. A relatively small lag table should be used due to the memory requirements of this approach. The code example uses a table length of 97, which means 388-bytes for the table per thread. Other larger prime number sized tables can be used to improve the period at the expense of memory utilisation. The input seed value is used to initialise a random number generator (RNG) on the host, which is then used to generate the seeds for the CUDA threads. The CUDA implementations of the lag table initialisation and uniform random number generator functions are essentially the same as on the CPU, only that ternary expressions, which can be optimised by the compiler, are used to avoid branches and array indexing is adapted so that global memory accesses can be coalesced as long as the threads of a half-warp always request a new random number at the same time.

Algorithm 4 CUDA implementation of Marsaglia's RNG that produces T independent streams of random numbers, where T is the number of threads. See Algorithm 5 for the CUDA kernel.

declare $T = 30720$ //thread count
declare $L = 97$ //lag table length
function RNG1(s)
  Input parameters: s is the initialisation seed.
  declare $S[T]$ //array of seeds
  initialise host RNG with s
  $S \leftarrow$ generate T random deviates on the host
  declare $S_d[T]$ in device memory
  copy $S_d \leftarrow S$
  declare $U_d[TL]$ in device mem. //lag tables
  declare $C_d[T]$ in device mem. //array of c values
  declare $I_d[T], J_d[T]$ in device mem. //indices
  do in parallel on the device using T threads:
    call KERNEL(S_d, U_d, C_d, I_d, J_d)

Algorithm 5 Algorithm 4 continued. The device kernel is the piece of code that executes on the GPU. The initialisation and uniform random number generator functions are essentially the same as on the CPU.

function KERNEL(S, U, C, I, J)
  declare $i \leftarrow$ thread ID queried from runtime
  $C[i] \leftarrow 362436.0/16777216.0$
  $I[i] \leftarrow L - 1$
  $J[i] \leftarrow L/3$
  declare $s \leftarrow S[i]$ //load the thread's seed
  initialise the thread's RNG using s
  generate random deviates when needed

A different approach has to be taken if a single sequence of random numbers is required. This approach, referred to as CUDA 2, only makes sense if most of the CUDA threads require the same number of random deviates and if giving the control back to the host before the next random number is needed does not come at a high cost or has to be done by the algorithm which consumes the random numbers anyway. The latter is necessary because this is the only way to synchronise across all CUDA threads. Algorithms 6 and 7 show how Marsaglia's algorithm can be adapted to generate random numbers in parallel using a single, large lag table. This approach is based on the fact that the window between the table indices i and j is shifted by one every time a new random deviate is generated and that they start with an offset of $\frac{2}{3}$ of the table size L. This means that $\frac{1}{3}L+1$ random numbers can be generated before index j reaches the starting index of i. It takes 3 iterations with either $\frac{1}{3}L$ or $\frac{1}{3}L+1$ threads each to generate L random numbers, as the table length is a prime and therefore odd. The only value that changes every time a random number is generated is c, but this is not a problem as all future values can be calculated as shown in the code fragments. However, the values for c calculated in this way and thus the resulting random numbers are slightly different to those generated in the usual fashion due to floating point rounding errors. This means that in order to get the same results when running a simulation with the same seed multiple times, it is necessary to use the same RNG implementation every time and not use this CUDA implementation once and the CPU implementation the next time.

Algorithm 6 CUDA implementation of Marsaglia's RNG that produces a single stream of random numbers using a large lag table. See Algorithm 7 for the CUDA kernel.

declare $L = 92153$ //lag table length
declare $T = L/3 + 1$ //thread count
declare $D = 7654321.0/16777216.0$
declare $M = 16777213.0/16777216.0$
function RNG1(s)
  Input parameters: s is the initialisation seed.
  declare $U[L]$ //the lag table
  $U \leftarrow$ initialise with s
  declare $U_d[L]$ in device mem. //the lag table
  copy $U_d \leftarrow U$
  declare $c \leftarrow 362436.0/16777216.0$
  while more random numbers required **do**
    //every iteration generates L random deviates
    declare $o \leftarrow 0$ //offset into the lag table
    declare $l \leftarrow L/3 + 1$ //update l table elements
    do in parallel on the device using T threads:
      call KERNEL(l, o, U_d, c)
    $o \leftarrow o + l$
    $l \leftarrow$ **round**$(L/3)$
    do in parallel on the device using T threads:
      call KERNEL(l, o, U_d, c)
    $o \leftarrow o + l$
    $l \leftarrow L/3$
    do in parallel on the device using T threads:
      call KERNEL(l, o, U_d, c)
    $c \leftarrow c - LD$ //update c for the next iteration
    $c \leftarrow c +$ **ceil**(**fabs**$(c)/M)M$
  end while

The host code initialises the lag table before it is copied to the device. It then calls the CUDA kernel 3 times with different offsets into the lag table, generating $L/3$ or $L/3+1$ deviates in each call for a total of L new random numbers. With a lag table of length 92153 and a thread block size of 64, 30720 CUDA threads are executed in each call, $2-3$ of which are

Algorithm 7 Algorithm 6 continued. The device kernel is the piece of code that executes on the GPU.

```
function KERNEL(l, o, U, c)
  declare t ← thread ID queried from runtime
  if t < l then
    declare i ← L − 1 − t − o //index i into lag table
    declare j ← L/3 − t − o //index j into lag table
    if j < 0 then
      j ← j + L
    end if
    c ← c − (t + o + 1)D //calculate c for thread t
    if c < 0.0 then
      c ← c + ceil(fabs(c)/M)M //until 0 ≤ c < 1
    end if
    declare r ← U[i] − U[j] //new random deviate
    if r < 0.0 then
      r ← r + 1.0
    end if
    U[i] ← r
    r ← r − c
    if r < 0.0 then
      r ← r + 1.0
    end if
    do something with r
  end if
```

unused.

Both CUDA implementations are mainly useful when the random numbers are consumed by other device functions, in which case they never have to be copied back to the host and often do not even have to be stored in global memory, but only exist in the local registers of the streaming multiprocessors. Lag table operations usually require global memory transactions, but if the conditions mentioned before are adhered, then all of these can be coalesced into 1 (approach 1) or $1-2$ (approach 2) transactions per half-warp.

3.5 Multi-GPU - CUDA & POSIX Threads

The multi-GPU version of our approach to implementing Marsaglia's RNG in CUDA is basically the same as its single-GPU counterpart. One pThreads is created for every CUDA capable device in the system. These threads are used to control the CUDA kernel preparation and execution on the device associated to them. Instead of having to compute T random deviates as seeds for the thread RNGs, the host now has to generate $T \times N$ seeds, where T is the number of threads per device and N is the number of devices. The multi-GPU implementation of Algorithm CUDA 2 does not distribute the lag-table across devices, but rather uses one lag-table per GPU.

3.6 Cell Processor - PS3

Implementing the lagged-Fibonacci generator on the Cell processor requires a certain deal of consideration. There are six separate SPEs each of which can process a vector for four elements synchronously. Vectors types are used to make full use of the SPEs processing capabilities. Thus for each iteration, each SPE will generate four random numbers (one for each element in the vector).

To ensure that unique random numbers are generated, each element in the vector of each SPE must have a unique lag table. Six SPEs with four elements per vector results in twenty-four lag tables. These lag tables are implemented as a single lag table of type `vector float` but each element of the vectors is initialised differently. Care should be taken when initialising these lag tables to make certain that the lag tables do not have correlated values and produce skewed results.

The lagged-Fibonacci generator algorithm has two conditional statements that affect variables of vector type. These conditional statements both take the form of `if( result < 0.0) result = result + 1.0;` (See Algorithm 1). As each element in the vector will have a different value depending on its unique lag table, different elements in the vector may need to take different branches.

Algorithm 8 Pseudo-code for Marsaglia Lagged-Fibonacci algorithm implemented on the CellBE using vectors.

```
  declare vector float u[97]
  initialise(u)
  declare i ← 96
  declare j ← 32
  declare c ← 362436.0/16777215.0
  declare d ← 7654321.0/16777215.0
  declare m ← 16777213.0/16777215.0
  function uniform()
    declare vector float zero ← spu_splats(0.0)
    declare vector float one ← spu_splats(1.0)
    declare vector float result ← u[i] - u[j]
    declare vector float plus1 ← result + one
    declare vector unsigned sel_mask ← result >
  zero
    result ← select(result, plus1, sel_mask)
    u[i] ← result
    i = i − 1
    if i == 0 then
      i ← 96
    end if
    j = j − 1
    if j == 0 then
      j ← 96
    end if
    c = c − d
    if c < 0 then
      c ← c + m
    end if
    result ← result−spu_splats(c)
    plus1 ← result + one
    sel_mask ← result > zero
    result ← select(result, plus1, sel_mask)
    return result
  end function
```

There are two ways of overcoming this issue. The first method is to extract the elements from the vector and process them individually. This method is not ideal as it does not use the vector processing ability of the cell, instead the `spu_sel` and `spu_cmpgt` instructions can be used.

The `spu_cmpgt` instruction will compare two vectors (greater than condition) and return another vector with the bits set to 1 if the condition is true and 0 if the condition is false. The comparison is performed in an element-wise manner so the bits can be different for each element. The `spu_sel` can then select values from two different values depending on the bits in a mask vector (obtained from the `spu_cmpgt` instruction).

Using these two instructions the conditional statement `if( result < 0.0) result = result + 1.0;` can be processed as vectors with different branches for each element. The pseudo-code for this process can be seen in Algorithm 8.

4 Multi-Platform Lagged-Fibonacci Performance Results

The implementations of the lagged-Fibonacci generators on different architectures have been tested by generating 24 billion random numbers and measuring the time taken. In the performance measures (See Table 1) the random numbers have not been used for any purpose as the only intention was to measure the generation time. This is obviously not useful in itself but it is assumed that any application generating random numbers such as these will make use of them on the same device as they were generated. Otherwise the random values can simply be written to memory and extracted from the device for use elsewhere.

Table 1: Comparison of the time taken to generate 24,000,000,000 random numbers using the lagged-Fibonacci generator on different hardware architectures. The CUDA measurements are done on a GeForce GTX295.

Device	Time (seconds)	Speed-up
CPU	256.45	1.0x
pThreads	66.72	3.8x
TBB	95.40	2.7x
Cell	23.60	10.9x
CUDA 1 (1 GPU)	8.56	30.0x
CUDA 1 (2 GPUs)	4.31	59.5x
CUDA 2 (1 GPU)	15.44	16.6x
CUDA 2 (2 GPUs)	8.33	30.8x

The platform we have used for all performance experiments except for the CellBE algorithms runs the Linux distribution Kubuntu 9.04 64-bit. It uses an Intel Core2 Quad CPU running at 2.66GHz with 8GB of DDR2-800 system memory and an NVIDIA GeForce GTX295 graphics card, which has 2 GPUs with 896MB of global memory each on board.

The platform used to run the CellBE implementations is a PlayStation 3 running Yellow Dog Linux 6.1. It uses a Cell processor running at 3.2GHz, which consists of 1 PowerPC Processor Element and 8 Synergistic Processor Elements, 6 of which are available to the developer. It has 256MB of system memory.

The results show that the concurrent implementations all perform well compared to the single-core CPU implementation. This comes as no surprise, as all threads and vector units execute independently from each another, using different lag tables and generating multiple streams of random numbers. The only exception to this is implementation CUDA 2, which generates a single stream of random numbers per GPU using a very large lag-table. The initial set-up time is insignificant compared to the time taken to generate 24 billion random numbers.

5 GPU Performance Results

In this section we focus on the GPU and report on the relative performance of different algorithms and the use of multiple GPUs. First we compare implementation CUDA 1 of the lagged-Fibonacci random number generator on different graphics devices, using 1, 2 or 3 GPUs and 3 different lag-table sizes to generate 24 billion random numbers in total. The results are given in Table 2. The algorithm scales almost linearly with the number of GPUs, which is not further surprising as the devices work independently and do not need to exchange any information. The GT200 series based devices show a significant performance drop when the lag-table size is increased, while the GTX480 is much less affected thanks to its improved memory hierarchy.

Table 2: Comparison of the time taken to generate 24,000,000,000 random numbers using implementation CUDA 1 of Marsaglia's lagged-Fibonacci RNG with lag-tables of size 97, 1021 and 4093 on various CUDA devices. The timing results are reported in seconds.

Device	GPUs	Lag-table size		
		97	1021	4093
GTX260	1	7.67	7.98	8.91
GTX260	2	3.87	4.16	5.08
GTX260	3	2.69	2.97	3.85
GTX295	1	8.56	8.82	9.50
GTX295	2	4.31	4.55	5.27
GTX480	1	4.21	4.36	4.31
GTX480	2	2.12	2.17	2.19

The second performance comparison puts our implementation of Marsaglia's RNG (CUDA 1) up against the algorithms Ran, Ranq1, Ranq2, Ranhash and Ranlim32 as described in Numerical Recipes 3rd edition (Press et al. 2007). The CUDA implementations of these RNG algorithms are straight forward and basically the same as the sequential CPU implementations. Each CUDA thread uses its own RNG instance and thus generates an independent stream of random numbers just like algorithm CUDA 1.

Two scenarios are used to compare the performance of these algorithms: **(a)** Generate 30.72 billion uniform deviates using 30720 threads and measure the execution time (lower is better); **(b)** Run an Ising simulation implemented in CUDA (Hawick et al. 2009) with 4096^2 cells for 16384 simulation steps and measure the hits per second (higher is better). The `Ranhash` algorithm is not well suited for the Ising simulation and has therefore not been used for those tests. Algorithm 9 describes how the different RNG implementations were tested for scenario **(a)**. The results are given in Table 3.

Algorithm 9 This pseudo-code describes how the performance of the different RNG algorithms was measured. A RNG running on the host is initialised with a single seed and then used to generate the seeds for the CUDA RNGs, which are stored in s, before the CUDA kernel is called. `rng_params` is a place-holder for all algorithm specific parameters. Every thread sums the random numbers that it generates and finally stores the value to global memory to avoid code from being removed during the compiler's optimisation phase.

```
tid ← thread ID queried from CUDA runtime
if tid < THREAD_COUNT then
  //init. the RNG stream with its individual seed
  initialise(tid, s[tid], rng_params)
  params ← load rng_params from global memory
  x ← 0.0
  for i ∈ {1, 2, 3, ..., 1000000} do
    x ← x+ generate_uniform(params)
  end for
  rng_params ← save params to global memory
  results[tid] ← x //store x to global memory
end if
```

Table 3: The performance results for two test scenarios using different RNG implementations. Lower results are better in the first scenario and higher results are better in the second one. A GTX295 has been used for these measurement.

Performance Results	Ran	Ranq2	Ranq1	Ranhash	Ranlim32	Marsaglia
Generate 10^6 uniform deviates per thread or 30.72 billion in total (seconds)	9.95	5.74	5.94	7.67	6.22	10.70
Ising simulation (10^9 hits per second)	2.15	3.16	3.12	N/A	3.23	2.26

6 Discussion

As indicated in table 3 the generator algorithms we have employed can be implemented so that they provide broadly similar performance on a typical GPU. Other things being equal we therefore would be drawn to choose a quality algorithm that has been well tested, and employed and reported in the research literature. The Lagged-Fibonacci algorithm is our favourite for this purpose, but it can be configured with various different lag-table sizes to improve the deviate quality.

The lag-table size that we have employed for algorithms like the Lagged-Fibonacci generator has a relatively marginal effect in slowing down the GPUs. A larger table obviously requires greater processing, but the memory utilisation itself is more likely of greater impact on simulations where GPU device memory is at a premium. This is particularly critical on the "gamer standard" GPUs we have employed in this work since current generation models typically have only around 1GByte of such device memory compared with "blade level" GPU products that have several times this amount.

The monotonic performance improvements we obtain on GPUs of increasing numbers of cores suggests an optimistic future for data parallelism on this architecture. AT the time of writing GPUs of approximately 2^9 cores are available and we believe the technology will readily support 2^9-2^{12} cores in the foreseeable future. We believe clusters using GPUs are already feasible, and it may even be beneficial to incorporate more than one GPU device per cluster node. This is certainly of use for applications where the work can be divided up into independent tasks. For other areas however, it may be more useful to allocate a specific accelerator device solely to producing random numbers. The CellBE architecture (if not this particular chip itself) shows some promise for that paradigm.

A particular application of interest for us is the Ising model(Onsager 1944) as described in (Hawick et al. 2009). It is notoriously difficult and computationally expensive to obtain high accuracy on critical parameters such as the critical temperature in the case of the three dimensional Ising system(Baillie et al. 1992). We are investigating how the critical temperature shifts when the underpinning lattice structure is changed according to a Watts-Strogatz re-wiring probability p(Hawick & James 2006). It is proving necessary to investigate p on logarithmic scales making the computational requirements even more severe. Using a portable generator such as the Lagged-Fibonacci algorithm that works well on all platforms available to us is very important to the computational feasibility of such numerical simulation problems. The Ising model is so important, that we have in fact used "Ising model Monte-Carlo Hits per second" as a performance metric of the random number generator algorithms – as presented in table 3.

We have not reported on FPGA(Danese et al. 2007) performance data nor on low-power commodity mobile devices such as ARM processor(Sloss 2010). It is possible to devote programmable array die-space to 64-bit operations and some ARMs can indeed perform 64 bit floating point. At least at the time of writing and perhaps for some few years to come, the present transient generation of devices will not necessarily be able to perform 64-bit operations at a commodity price regime and therefore the issues we have discussed about portability on 32-bit devices will remain valid.

7 Conclusions and Future Work

We have explored the portability and performance of various random number generators on different accelerator devices using variety of parallel programming frameworks. The data-parallelism of the GPU architecture is particularly attractive for the Monte Carlo work we have focused on. While random number generation is surely still a area where the "horses for courses" argument applies, depending upon application context, we do believe the Marsaglia lagged-Fibonacci generator with suitable lag-table size is still a worthy portable candidate suited for use in quality Monte Carlo studies.

For future work we believe hybrid processor architectures such as the CellBE are interesting and will offer good specialist pipelining capabilities – such as generating random numbers. However at the time of writing we believe the software toolset available to help program such devices places quite high burdens on the applications programmer. While CUDA is not totally trivial it is certainly more application friendly, and we do expect its programming models to propagate further into systems like OpenCL.

OpenCL holds some promise for portability although we have not reported on detailed performance data since at present the OpenCL platforms available to us are far out stripped by CUDA. We do expect this situation will change as more vendors take OpenCL up. However, since RNGs tend to make use of low-level computational facilities such as bit operations, and would usually be written to take advantage of any vector or pipeline facilities available, it is not clear whether OpenCL level portability will be enough.

In summary, we believe that at the time of writing, the Marsaglia Lagged-Fibonacci algorithm operating using single precision floating point is both portable across the devices we discuss and is readily implementable using the software technologies available - with suitable care and attention. The performance attainable multi-GPU data-parallel threads within the context of multiple conventional threads or processes of a multi-gpu cluster is particularly encouraging. This is our platform of choice for our current Monte-Carlo work, with the caveat of requiring suitable care and caution about seed initialisation and effective periodicity issues.

References

Bader, D. A., Chandramowlishwaran, A. & Agarwal, V. (2008), On the design of fast pseudo-random number generators for the cell broadband engine and an application to risk analysis, *in* 'Proc. 37th IEEE Int. Conf on Parallel Processing', pp. 520–527.

Baillie, C., Gupta, R., Hawick, K. & Pawley, G. (1992), 'Monte-Carlo Renormalisation Group Study of the Three-Dimensional Ising Model', *Phys.Rev.B* **45**, 10438–10453.

Binder, K. & Heermann, D. W. (1997), *Monte Carlo Simulation in Statistical Physics*, Springer-Verlag.

Brent, R. P. (1997), A fast vectorized implementation of wallace's normal random number generator, Technical report, Australian National University.

BSD (1993), *Random - Unix Manual Pages*.

Coddington, P. D. (1994), 'Analysis of random number generators using monte carlo simulation', *J. Mod. Physics C* **5**(3), 547–560.

Coddington, P. D. & Ko, S. H. (1998), Techniques for empirical testing of parallel random number generators, *in* 'Proc. International Conference on Supercomputing (ICS'98'. DHPC-025.

Coddington, P., Mathew, J. & Hawick, K. (1999), Interfaces and implementations of random number generators for java grande applications, *in* 'Proc. High Performance Computing and Networks (HPCN), Europe 99, Amsterdam'.

Coddington, P. & Newall, A. (2004), Japara - a java parallel random number generator library for high-performance computing, Technical Report DHPC-144, The University of Adelaide.

Cuccaro, S., Mascagni, M. & Pryor, D. (1995), Techniques for testing the quality of parallel pseudo-random number generators, *in* 'Proc. of the 7th SIAM Conf. on Parallel Processing for Scientific Computing,', SIAM, Philadelphia, USA, pp. 279–284.

Danese, G., Leporati, F., Bera, M., Giachero, M., Nazzicari, N. & Spelgatti, A. (2007), 'An accelerator for physics simulations', *Computing in Science and Engineering* **9**(5), 16–25.

Deng, L.-Y. & Xu, H. (2003), 'A system of high-dimensional, efficient, long-cycle and portable uniform random number generators', *ACM TOMACS* **14**(4), 299–309.

Giles, M. (2009), Notes on CUDA implementation of random number genertors. Oxford University.

Hawick, K. A. & James, H. A. (2006), Ising model scaling behaviour on z-preserving small-world networks, Technical Report arXiv.org Condensed Matter: cond-mat/0611763, Information and Mathematical Sciences, Massey University.

Hawick, K., Leist, A. & Playne, D. (2009), Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs, Technical Report CSTN-093, Computer Science, Massey University. To appear in Int. J. Parallel Programming (2010).

Hormann, W. (1993), 'The transformed rejection method for generating poisson random variables', *Insurance: Mathematics and Economics* **12**(1), 39–45.

ID Quantique White Paper (2010), Random Number Generation Using Quantum Physics, Technical Report Version 3.0, ID Quantique SA, Switzerland. QUANTIS.

Johnsonbaugh, R. (2001), *Discrete Mathematics*, number ISBN 0-13-089008-1, 5th edn, Prentice Hall.

Khronos Group (2008), 'OpenCL - Open Compute Language'.
URL: *http://www.khronos.org/opencl/*

Knuth, D. (1997), *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, 3rd edn, Addison-Wesley.

Langdon, W. (2009), A Fast High Quality Pseudo Random Number Generator for nVidia CUDA, *in* 'Proc. ACM GECCO'09'.

L'Ecuyer, P. (2001), Software for uniform random number generation: distinguishing the good and the bad, *in* 'Proc. 2001 Winter Simulation Conference', Vol. 2, pp. 95–105.

Marsaglia, G. (1984), A Current view of random number generators, *in* 'Computer science and statistics: 16th symposium on the interface', Atlanta. Keynote address.

Marsaglia, G., Zaman, A. & Tsang, W. W. (1987), 'Toward a universal random number generator', *Statistics and Probability Letters* **9**(1), 35–39. Florida State preprint.

Matsumoto, M. & Nishimura, T. (1998), 'Mersenne twistor: A 623-diminsionally equidistributed uniform pseudorandom number generator', *ACM Transactions on Modeling and Computer Simulation* **8 No 1.**, 3–30.

Newell, A. (2003), Parallel random number generators in java, Master's thesis, Computer Science, Adelaide University.

Onsager, L. (1944), 'Crystal Statistics I. Two-Dimensional Model with an Order-Disorder Transition', *Phys.Rev.* **65**(3), 117–149.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. & Flannery, B. P. (2007), *Numerical Recipes - The Art of Scientific Computing*, third edn, Cambridge. ISBN 978-0-521-88407-5.

Schneier, B. (1996), *Applied Cryptography*, second edn, Wiley. ISBN 0-471-11709-9.

Sloss, A. N. (2010), *ARM System Developer's Guide: Designing and Optimizing System Software*, Elsevier. ISBN 978-0080-490-496.