An Exploration of General Purpose Programming on GPUs

MSc Dissertation

Michael Fergus McCann

A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Science under the supervision of Dr. Neil Hurley.

School of Computer Science and Informatics

University College Dublin

April 2011

Abstract

The recent emergence of simplified models for general purpose GPU programming has led to an explosion in the popularity of GPU accelerated applications development. An overview of this phenomenon is presented along with a GPU based parallel implementation of the well known RANMAR pseudo random number generator. The parallel RANMAR implementation is shown to exhibit up to a 5 fold speed up over the sequential version on one system tested. Exhaustive statistical tests were run on the numbers produced and a potential weakness in at least two common implementations of the double precision RANMAR are discussed. The implementation is integrated with Corsika (the widely used, FORTRAN based, high-energy cosmic radiation interaction simulation software). Finally, the suitability of generating random numbers using GPU hardware is discussed.

I would like to thank Dr. John Quinn of the UCD School of Physics for providing the real world problem that an investigation such as this requires. I would also like to thank him for providing a GPU enabled hardware environment and an invaluable explanation of the physics behind the simulations that this thesis aimed to (GPU) accelerate.

I would also like to thank Dr. Neil Hurley, my project supervisor, for his advice and encouragement for the duration of this project.

Table of Contents

1 Introduction

1.1 GPGPU Background

General purpose programming on GPUs began around 2001 with the arrival of the first GPUs with programmable graphics pipelines[1]. Researchers were quick to spot the potential of using the raw computational power (figure 1) of the GPU in solving non-graphics problems - but the original programming models were unwieldy. In essence, early GPGPU[2] programming required the recasting of problems in terms of GPU rendering pipeline stages, such as vertex or fragment shading. This was not only inconvenient but it meant that the set of problems that could be practically solved using GPUs was greatly limited. This in turn meant that for several years, GPGPU programming remained the preserve of determined researchers only.

This picture has been rapidly changing due to a number of parallel developments. Firstly, in recent years simpler GPU programming models have emerged, including: Close To Metal (2006), CUDA (2007) and OpenCL (2008). These models now allow programmers to concentrate on the problem domain under consideration by treating the GPU as a general purpose parallel SIMD processor and using familiar high level language bindings such as C or Python. Secondly, over the same period, GPU hardware has advanced broadly in line with Moore's law [3] [18] so much so that it is now possible to buy reasonably priced consumer grade GPU hardware providing teraflop range (peak) performance (e.g.: NVIDIA GeForce GTX 480, Radeon HD 5870).

Figure 1. An NVIDIA illustration [25] of the potential computational power of the CPU by showing the relative ALU densities between CPUs and GPUs

[1] NVIDIA's GeForce 3 in 2001 and ATI's Radeon 9700 in 2003

[2] General Purpose Programming on Graphics Processing Units (GPGPU)

Thirdly, there has been on ongoing computing technology trend away from sequential programming environments to multi- and many-core computer systems [7], [8], [9] and [10]. This has further driven the uptake of general purpose GPU usage.

The shift away from single core commodity systems began around 2003 when it became harder to fabricate processors that worked reliably at clock speeds in excess of 3.5GHz. Power dissipation also became an issue and ultimately manufacturers decided that the marginal gains achieved through hard won clock speed increases were not cost efficient. As others soon after observed, "The major chip manufacturers have, for the time being simply given up trying to make processors run faster" [10]. This marked the beginning of what has been described as two separate microprocessor design trajectories: the multi-core and many-core trajectories [9]. The multi-core trajectory, aimed at optimising sequential programs, involved casting multiple processor cores on the same chip, each one capable of running its own instruction stream. The many-core systems (such as GPUs) aimed at optimising parallel programs by providing a far greater number of smaller cores, capable of running only one instruction stream. These systems were also designed to specialise at performing data parallel compute intensive applications and so dedicated more of their of chip transistors to computation and rather less to flow control and caching as depicted in figure 1.

In summary, the simplified programming models, the performance characteristics and the highly parallel nature of many-core GPU hardware has been driving the increased prevalence of GPU programming in recent years. While it cannot be said that GPU programming is an area familiar to most working software engineers, the current abundance of published material (e.g.: [3], [4], [5], [16] and the series starting at [6]) related to general purpose programming on GPUs as well as the wide application of GPUs to non graphics problem domains [15] suggest that GPU programming is ready to become mainstream.

1.2 Project Outline and Goals

Primarily, this is a project about general purpose programming on GPUs. Reports abound of orders-of-magnitude speed ups being achieved by porting sequential algorithms to commodity parallel GPU hardware (for some examples see [17]). Motivated by such reports, this project aimed to investigate a real world problem to see if GPU acceleration could provide a cost effective speedup to a particularly lengthy computation. Specifically, a UCD School of Physics research group performing Monte-Carlo simulations of cosmic radiation interactions with the Earth's atmosphere (using Corsika [21] and [22]) found that up to 80% of the simulation time was spent generating random numbers. This project implemented a GPU accelerated version of their pseudo random number generator – the RANMAR as described by Marsaglia in [12] and in a slightly modified form by James in [13].

The RANMAR implementation presented here is original in the sense that no reference could be found to any existing CUDA implementation (NVIDIA's programmable GPU architecture) and while an implementation for ATI graphics cards was found [14] and [23], the approach described here is more sophisticated. The adaptation of the algorithm

for parallel processing and the development process including all attendant challenges are presented along with a statistical validation of the random numbers sequences produced. Next, a performance comparison between the new GPU implementation and a sequential, CPU based implementation are described in detail. Finally, and for further comparison purposes, the statistical validation process is applied to the existing double precision RANMAR algorithm found in the Corsika simulation package. Some suspicions about the approach taken in Corsika (and also in the CERN program library) are presented.

Based on the research and implementation experiences conclusions are drawn about the relative accessibility and applicability of using graphics processors for solving non graphics problems.

2 The Problem Domain

While the introductory paragraphs have already made clear that the implementation aspects of this thesis involve GPU accelerated random number generation, it is important to understand the motivations behind the choice of problem. In a way, this project began life as a solution looking for a problem, that is, it began with an awareness of the potential power of GPUs and a desire to solve a real world problem. The high energy astrophysics research group in the U.C.D School of Physics provided such a problem and at the same time, a collaboration opportunity. In order to fully understand this project's motivations, the following paragraphs explain the motivations of the astrophysics research group and their desire for any possible acceleration of their cosmic radiation Monte-Carlo simulations.

2.1 The Role of Monte-Carlo Simulations in Cosmic Ray and Gamma Ray Astronomy

2.1.1 Introduction

The Earth is continually bombarded by high-energy particles (known as cosmic rays) and photons (gamma rays), with energies orders of magnitude greater than those of CERN's LHC[3] experiments. The quest to determine the sources of this cosmic radiation is almost 100 years old – dating since its discovery by Victor Hess in 1912. While the Earth's atmosphere provides shielding which protects the surface from this harmful radiation, it also provides an indirect mechanism to detect and study the very high energy cosmic rays and gamma rays by detecting and studying the secondary showers of relativistic particles ("extensive air showers") that are created when the high energy cosmic radiation interacts with the earth's atmosphere. In the last two decades a new branch of astronomy has emerged, that of TeV[4] gamma ray astronomy – this research field involves the detection

[3] Large Hadron Collider – interested readers can get an overview of its history and the kinds of experiments performed at CERN at http://en.wikipedia.org/wiki/Large_Hadron_Collider

[4] TeV = tera-electronvolt

of the secondary particle showers in the atmosphere, and the development of techniques to discriminate gamma-ray induced showers from cosmic ray induced showers. There are currently three state of the art TeV observatories in operation (VERITAS in Arizona, MAGIC in La Palma and HESS in Namibia). Between them these operations have detected over 110 astronomical sources of TeV gamma rays and Monte Carlo simulations of extensive air showers are critical to the success of these observatories.

2.1.2 Extensive Air Showers and TeV Gamma-ray Astronomy

When a gamma ray of sufficiently high energy (>1~ GeV, about 1,000,000,000 times more energetic than a visible photon) interacts with the atmosphere it induces an electromagnetic cascade, that is, a shower of hundreds/thousands of electron-positron pairs and gamma rays. The development of such a shower of particles in the atmosphere is determined by the physics of pair production (where gamma rays interact with matter to transform into electron-positron pairs) and Bremsstrahlung (where energetic particles interact with matter to emit high-energy photons) and is a statistical process. Cosmic ray particles also induce air showers, but these are quite different in structure due to different physical mechanisms involved. Even though the shower of particles dies out before it reaches ground, an air shower can be detected via the Cherenkov radiation [20] that is emitted as the particles travel relativistically through the atmosphere. The early attempts at detecting gamma rays from these Cherenkov flashes of light form air showers proved very difficult as the small gamma-ray signal is overwhelmed by the background of cosmic rays. The key breakthrough in the field of TeV gamma ray astronomy came with the development of 'imaging', where a photomultiplier tube (PMT) camera is used to record an image of the shower development in the atmosphere. The images are analysed off-line to produce a set of parameters (e.g.: length, width etc.) that characterise them. Monte-Carlo simulations of thousands of gamma ray and cosmic ray induced air showers are used to derive 'cuts', selection criteria that distinguish between gamma-ray and cosmic ray induced showers. This technique was used for the first detection of a TeV gamma ray source, the Crab Nebula [19] and since then the field has flourished. Modern TeV observatories use arrays of imaging telescopes, and as no calibrated source of TeV gamma rays exists in nature, all information derived about the gamma-ray emission from astronomical sources is totally dependent on Monte-Carlo simulations of:

- The air-showers
- Propagation of Cherenkov photons through the atmosphere
- Reflection off the mirrors and detection by the photomultiplier tubes.

The response of the trigger and digitisation systems must also be included in these simulations.

2.2 UCD High Energy Astrophysics Group and VERITAS

The Very Radiation Imaging Telescope Array System (VERITAS) is an array of four 12 metre diameter imaging atmospheric Cherenkov telescopes for TeV gamma-ray astronomy. It is located in southern Arizona, and has been in full scientific operation since autumn 2007. The high energy astrophysics group at UCD is a member of the VERITAS collaboration, which has 94 members from 24 institutions in the USA,

Canada, UK and Ireland. Each telescope contains a 499-pixel photomultiplier-tube camera and a three-level trigger system is used to determine whether events should be recorded. Images from the PMT cameras are digitised by a 500 MSPS flash ADC system and analysed off-line. Within the collaboration, the Monte Carlo Simulation Working Group provide simulations of the response of the VERITAS to gamma-ray and Cosmic Ray-induced Air Showers, which are vital to understanding the performance of the array and producing scientific results. Monte Carlo simulations of air showers and instrument response are performed on clusters of computers and take months to run therefore, the implementation of mechanisms to speed up this process is highly desired by the collaboration because instrumental or atmospheric changes (e.g.: the atmospheric transmission in Arizona in 2008 changed dramatically due to forest fires) require new Monte Carlo simulations to be generated which holds up image analysis.

2.3 CORSIKA

The first step in identifying ways to potentially speed up simulations was to look at the simulation software. Corsika [21], [22] is the software package used by the VERITAS collaboration to perform their simulations. It is a very widely used, open source, mainly FORTRAN based, cosmic radiation interactions simulation package. By profiling simulation runs, the UCD astrophysics group have determined that the random number generation aspect of the Monte-Carlo simulations tends to dominate the run-time. Therefore, RANMAR [12], the random number generator used by Corsika was the chosen area for potential GPU acceleration.

2.4 The RANMAR Pseudo Random Number Generator

The RANMAR pseudo random number generator of Marsaglia, Zaman and Tsang [12] consists of a combination of a lagged Fibonacci generator (LFG) and a simple arithmetic sequence. The general form of an LFG, describing how to generate the *rth* element of the random number sequence, for a given binary operation $\bullet$ and lags i and j is given by:

$$x_r = x_{r-i} \bullet x_{r-j}$$

For RANMAR, as originally described, we have the following specific LFG:

$$x \bullet y = \{ if\ \ x \geq y\ \ then\, x - y\,,\ else\ \ x - y + 1 \}$$

$$i = 97\,,\ \ j = 33$$

When used with 24 bit fractions, this sequence is equivalent in period and structure to an integer LFG with lags i and j with operation subtraction modulus 2^{24}. This gives a theoretical maximum period of $(2^{24} - 1) \times 2^{96}$. Marsaglia et al point out that this LFG, assuming adequate assignment of the initial 97 seed values, provides a pseudo random number sequence that is almost good enough (i.e.: has a long period and appears sufficiently random) but that it fails the "birthday spacing test". For this reason, the

RANMAR combines the LFG with a simple arithmetic sequence for the prime modulus $2^{24}-3=16777213$. The sequence is defined as follows. For a sequence with a current value c_n the next value in the sequence c_{n+1} is given by:

$$c_{n+1} = c_n \circ d$$

where:

$$c \circ d = \{ if\ c \geq d\ \ then\ c-d\ , else\ c-d+16777213/16777216 \}$$

$$c_1 = 362436/16777216$$

$$d = 7654321/16777216$$

The combination of the LFG and the arithmetic sequence is then performed such that for an LFG sequence and arithmetic sequence respectively given by:

$$x_1, x_2, x_3, \ldots x_n$$

$$c_1, c_2, c_3, \ldots c_n$$

We produce the final (uniformly distributed) random number sequence:

$$U_1, U_2, U_3, \ldots \quad \text{where } U_n = x_n \bullet c_n$$

The assignment of the initial 97 values for the LFG are of paramount importance and rather than impose that responsibility on the end user, Marsaglia et al provide an algorithm to generate their values from just 4 user provided seed values while James [13] while working for CERN, modified it very slightly, reducing the number of required seeds to 2. The algorithm, in an effort to remain machine portable, generates these values on a bit by bit basis, up to the supported 24 bits. James further draws attention to a very useful feature of the RANMAR which is the ease with which a user can generate multiple independent pseudo random number sequences. Considering his system of providing two seeds, there are 31329 possibilities for the first seed and 30082 possibilities for the second seed, with each combination (>942 million) giving rise to an independent, non-overlapping sequence with an average period of 10^{30}.

2.4.1 Extension to Double Precision

The RANMAR as just described generates floating point numbers of single precision, that is, numbers using only a 24 bit mantissa (any other available mantissa bits would remain zero). This is because the initial values in the LFG and the elements of the arithmetic sequence are all initialised with 24 bits and consequently all additive and subtractive combinations of these values as per the RANMAR algorithm will still only have 24 bits. The implementation used in Corsika (and in its previous incarnation written by James as part of the CERN program library [24]) generates double precision floating

point numbers, using a 48 bit mantissa. The change to support this was trivial, simply requiring the initialisation routine to generate 48 bits fractions for the initial 97 LFG values instead of 24 bit fractions. Given that the LFG initialisation algorithm generates each bit value individually, all that was required was to loop 48 times instead of 24 times.

It is noteworthy that both the CERN program library implementation and the Corsika implementation decided that converting the LFG component of RANMAR to use 48 bits was sufficient. The arithmetic sequence remained 24 bit. This means that the lower order 24 bits of elements of the LFG always remain unperturbed when the LFG is combined with the arithmetic sequence. Given the concerns expressed by Marsaglia et al about the randomness of the LFG on its own, it is possibly of concern that at least two widely used 48 bit RANMAR implementations only apply the arithmetic sequence to the high order 24 bits of the LFG. Further investigations into this aspect of the double precision implementation are presented later.

3 GPU Programming and CUDA

3.1 Background

As discussed in the introduction, GPU programming has been greatly simplified since the introduction of programming models that present the GPU as a general purpose SIMD[5] like processor with a rich API. While some GPU software architectures have come and gone – at the time of writing there are really only two options available: CUDA and OpenCL. CUDA is the NVIDIA proprietary model that works with NVIDIA GPUs only[6]. OpenCL is an open GPU programming framework originally developed by Apple but currently belonging to the collaborative standards organisation – the Khronos group. Despite OpenCL being standardised and supported by both NVIDIA and AMD/ATI GPU drivers it has not yet achieved the same popularity as CUDA. This is primarily because CUDA has been around longer and is therefore mature, stable and trusted, especially among the HPC community. For these reasons and because of the easy availability of documentation and an internet support ecosystem, this project decided to perform its implementation of the RANMAR using CUDA.

[5] Although the CUDA model is frequently described as SIMD, it is, in reality SPMD, which allows different instructions from a single instruction stream to be executed concurrently on different processing units.

[6] CUDA has recently added support for x86 CPUs, exploiting available multi-cores or SSE. It will still not support AMD/ATI GPUs however.

3.2 The CUDA Model

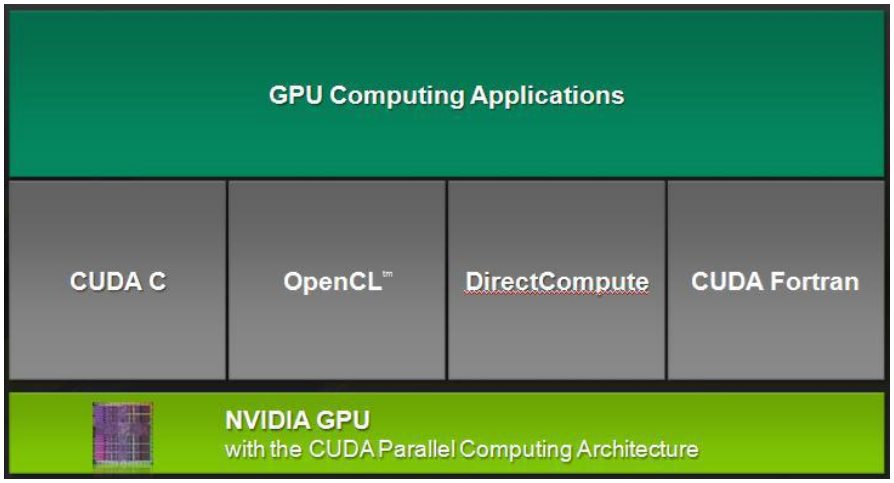


Figure 2. The CUDA Architecture (taken from [25])

The CUDA architecture supports the development of GPU applications using a choice of languages or APIs. CUDA C, which is based on C (with some small extensions) is of most interest in this project as it is used in the RANMAR implementation, but Fortran would also have been possible as would use of OpenCL (also based on C). The CUDA programming model can be understood at a high level by considering the areas outlined in the following sections:

3.2.1 The CUDA C Coding and Compilation Model

A CUDA C program utilising a GPU will consist of C code to be run on the CPU and C code to be run on the GPU. The code for both can reside in the same translation unit or they can be separated according to developer preference. The CPU (host) code provides the program entry point and overall program logic while the GPU (device) code consists of one or many "kernels" which are C functions that can be run on the GPU. The overall program execution must begin and end on the CPU, and the CPU code is responsible for making calls to the GPU kernels. Kernels may call other kernels, but they cannot make calls back to the CPU.

The extensions to the C language to enable this model are minimal. A device kernel is differentiated from a normal host function by adding a simple prefix ("__global__") to its signature. Host code calls kernel functions using a new triple-chevron notation that specifies the kernel name and the number of kernel instances that are to be concurrently run (see sample code in figure 3).

When ready to compile, the CUDA C compiler separates the device code from the host code and generates either architecture specific binary files (cubin format) or architecture independent PTX assembler code that can be run on the GPU. The compiler also inserts kernel calls into the host code binding the two code domains. Once the host and device code are separated, the host code is compiled with the normal native C/C++ compiler and linked against the CUDA runtime library.

```
// Kernel definition
__global__ void myKernel(double* x, double* y, double* z)
{
    int tid = threadIdx.x;
    z[tid] = x[tid] + y[tid];
}
// Usual CPU entry point
int main()
{
    ...
    // Execute myKernel on GPU with 100 threads
    myKernel<<<1, 100>>>(a, b, c);
}
```

Figure 3. Sample code showing a kernel that adds 2 vectors and its invocation

3.2.2 The CUDA Task and Data Parallelisation Model

As can be seen from the sample code in figure 3, the kernel invocation call specifies the number of threads to run. In that example, 100 threads were specified (while the other argument specifies that there should be 1 thread block). A form of task parallelisation is achieved by specifying a kernel be run in multiple blocks, but the tasks in each block must be independent because there are no synchronisation primitives available across blocks. For each block, the number of required threads is specified, and synchronisation mechanisms are available between threads in the block. This allows for the implementation of solutions to typical data parallel problems.

Kernel execution instances have access to their individual thread index via the build in variable threadIdx. They also have access to the index of the thread block in which they are running via the built in blockIdx variable. This allows kernel threads, in classic data parallel fashion, to access the specific area of a problem for which they have responsibility.

When the RANMAR implementation is explained in the next chapter, the use of both multiple blocks and multiple threads are used in order to maximise GPU use and in turn RANMAR performance.

3.2.3 The CUDA Memory Model

One of the largest performance related GPU programming factors is the pattern and types of memory usage. The following briefly explains the CUDA memory hierarchy because an understanding of this is of importance in the RANMAR implementation.

3.2.3.1 CPU RAM

In the context of a CUDA program, CPU RAM is available to the GPU in one of two ways:

- Data residing in CPU RAM is explicitly copied to GPU global memory by host code before making kernel calls. This is the most common pattern.
- Host code can explicitly allocate page-locked CPU memory which can then be accessed from the GPU using DMA.

3.2.3.2 GPU Global memory

This is the largest area of NVIDIA GPU memory - it is typically implemented with off-chip DRAM and it is the slowest GPU memory (hundreds of clock cycles). Any copy issued from host code will copy data into this area of memory. Global memory is readable and writable by all thread blocks and threads running on the GPU. Data in global memory persists across kernel calls and so can be used to maintain state with a lifetime exceeding a kernel execution.

3.2.3.3 GPU Local memory

Local memory is somewhat of a misnomer as it is actually global memory. Device code with automatic array variables (called local memory) will actually use the same DRAM global memory as per section 3.2.3.2, except the scope of the variable is thread local and the lifetime is that of the kernel invocation.

3.2.3.4 GPU Shared memory

Shared memory is low latency on-chip memory that is shared among all threads in the same thread block. It therefore offers a performance viable mechanism for collaboration between threads in a thread block. Variables in shared memory persist only for the duration of the kernel execution and as already explained, have thread block scope.

3.2.3.5 GPU registers

GPU register memory is also low latency on-chip memory. It has thread scope and it persists for the duration of the kernel only. All automatic scalar variables in kernels are placed in registers by default.

3.2.4 The CUDA Program Flow Model

While CUDA programming may vary considerably according to problem complexity, the most common program flow is as follows:

1. Launch host program
2. Host code allocates memory on GPU device using CUDA allocation primitives.
3. Host code copies any data required by the GPU from CPU RAM to GPU global memory (although as already discussed, DMA from the GPU is also possible).
4. The host invokes a kernel specifying number of thread blocks and the number of threads in each block.

5. The device kernel executes, according to its algorithm, potentially using shared memory, constant memory or texture memory[7].

6. The device kernel makes its results available in GPU global memory before returning control back to CPU.

7. The host code copies result data from GPU global memory to CPU RAM.

The GPU RANMAR implementation presented in this thesis matches this standard program flow matches almost exactly.

4 Parallel RANMAR Using a CUDA GPU

In order to implement RANMAR for a CUDA GPU environment the sequential algorithm described by Marsaglia et al must be parallelised in some fashion. There would be little point in replicating the algorithm exactly on the GPU because that would be akin to treating the GPU as another (slower) CPU. The following sections describe the approach that was taken and detail the challenges and limitations encountered.

4.1 Parallel RANMAR Design

There are three mechanisms for parallelising random number generators discussed by Coddington in [26]. These are:

1. **Leapfrog**: This is a mechanism where the random number sequence generated is the same no matter how many processors are employed. It is useful where programs utilising a random number sequence wish to get the same numbers every test run despite the number of processors potentially varying. Note that it is also useful in current context, when implementing a new parallel algorithm and where one wishes to validate the algorithm by comparing the generated values with those generated by a reference sequential implementation. The method works in a parallel system of N processors, where each processor generates every N^{th} sequence element. This can also be expressed more succinctly by stating that the P^{th} processor generates the sub-sequence:

$$X_P, X_{P+N}, X_{P+2N}, \ldots.$$

Of course this approach is only suitable when it is possible for processors to skip ahead in the random number sequence to those numbers for which they have responsibility. As we shall see an LFG with sufficient lags and an arithmetic sequence are candidates in this regard.

[7] Constant and texture memory are other off-chip memory options - designed for specialised access patterns. For a detailed explanation on these and all CUDA related issues, refer to [1] and [2]

2. **Sequence Splitting**: This is a mechanism similar to the leapfrog except where each processor generates a block of contiguous sequence elements before skipping ahead and generating its next block. In a system of N processors and for a block size of L, the P^{th} processor generates the sub-sequences:

$$X_{PL}, X_{PL+1}, X_{PL+2}, ..., X_{PL+L-1}, ...X_{2PL}, ...$$

   As with the leapfrog method, sequence splitting requires a number generator that allows processors to skip ahead in the random number sequence. This method shares the same advantage as the leapfrog in that the same sequence is generated regardless of the number of processors in the system.

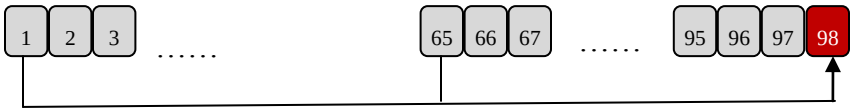
3. **Independent Sequences**: This is perhaps the most intuitive approach because it simply involves using multiple independently seeded random number sequences presented to the caller as a single sequence. Each processor either populates some target random number array in a fashion similar to the leapfrog method (that is every N^{th} element) or it populates the target array in blocks similar to the sequence splitting method. Ultimately, because an independent sequence is generated by each processor, the combined random number sequence will depend both on the number of processors in the system and the manner in which the sub-sequences are combined.

The parallelisation of RANMAR was achieved using both the leap frog and independent sequences methods. The following sections describe the specific nature of the design.

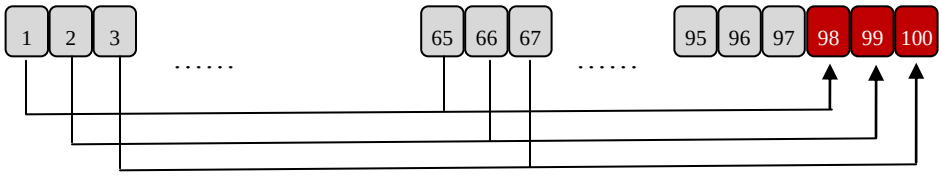
4.1.1 RANMAR Parallelisation within a Single Sequence (Leapfrog)

In this section the design employed to realise a leapfrog style parallelisation of the RANMAR is presented. It is noteworthy that no example of such an implementation could be found during the research phase of this project and therefore it is this aspect of the parallel RANMAR that differentiates it from the ATI implementation found in [14] and [23].

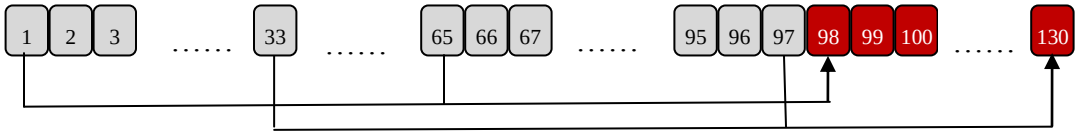
For any RANMAR sequence's current state, it is always possible to generate a certain number of subsequent random numbers concurrently. This is because of the specific properties of the RANMAR generator - in particular, the fact that it is made up of an LFG with lags convenient for parallelisation and that the other component is a simple parallelisable arithmetic sequence. Taking the LFG component first of all - with lags of 97 and 33, it is possible to generate the next 33 elements of the LFG component concurrently because there is no dependency between these next 33 values and the values needed to compute them. In order to see this, consider the initial state of the LFG, which is a buffer of size 97 which can be thought of as holding the first 97 values. In order to calculate the 98th value, the values at position 1 and position 65 must be combined (i.e.: using the particular lags as mandated by the RANMAR definition, 98 – 97 and 98 – 33). It is easy then to see that there would be no reason why the 99th and 100th element in this sequence could not be calculated at the same time as the 98th because there are no inter-dependencies in these calculations.



(i) The 98th element is computed by combining the 1st and the 65th element



(ii) The 98th, 99th, 100th ... element computations have no inter-dependencies



(iii) The possibilities for concurrent computation extend to the 130th element

Figure 4. The parallelisation of a single RANMAR sequence

In fact the parallelisation in the LFG can be extended all the way up to the 130th element (which depends on the 97th and 33rd elements). However, this exhausts the opportunities for parallelising the LFG because generating the 131st element in parallel would require the 98th element which may or may not be available. In implementation terms, the buffer illustrated in figure 4 is actually circular so that the newly computed 98th element of the sequence is placed in the array in position 1 and the 99th value is placed in position 2 and so on. The essential aspect is that at any time, the next 33 elements in the LFG sequence can be computed simultaneously and that an individual thread could be assigned to each of these computations.

The second component of the RANMAR is the arithmetic sequence. While it is described by Marsaglia et al [12] as a "simple arithmetic sequence for the prime modulus $2^{24}-3=16777213$" it actually presents some difficulties for parallelisation. The specific parameters characterising the sequence have already been explained and a sequential realisation is trivial to implement. This is because in order to generate the next element in the sequence all that is required is a simple manipulation of the current element. For a parallel implementation however, we need a method such that, for any current element c_n we can generate some later element c_{n+x}. Initial attempts to find such a method using floating point arithmetic (as in the sequential version) proved fruitless because they necessarily involved floating point multiplication and modulus operations

which introduced the usual floating point inaccuracies[8]. Instead it was necessary to use integer arithmetic for the sequence calculations (none of which require a division) and to divide by 2^{24} at the end to convert back to a 24 bit fraction. For the integer calculations, all values are the same as per the RANMAR specification, except they are shifted 24 binary decimal places to remove their fractional part. The eventual solution to the parallelisation of the arithmetic sequence is described in the algorithm presented in figure 5.

- The current thread identifies the current sequence value (that is the value c_n) and the offset to the element that it is expected to compute (that is x)
- The current thread then performs the following on c_n:
    - Subtracts $x.d$ (equivalent to x subtractions)
    - Performs the modulus $2^{24}-3$ operation
    - If the value is less than zero, add $2^{24}-3$
- Finally the answer is converted to floating point and divided by 2^{24}

Figure 5. Algorithm for the first pass of the (single precision) RANMAR arithmetic sequence

Note that the algorithm presented describes the first pass calculation. To understand the complete algorithm, say we have 32 threads (as we shall see later, this is in fact the chosen number of threads) and we wish for each thread to calculate 1 of the next 32 elements of the sequence simultaneously, such that thread 1 computes the next value, thread 2 computes the next plus one and so on. When the generation process starts, these first 32 values constitute the first pass whereby all threads calculate relative to some common starting value (c_n in the algorithm in figure 5). For subsequent elements however each thread can calculate its next assigned element relative to the value it just computed. In other words, it works by always calculating the value of the sequence 32 positions from the thread's current position (assuming there are 32 threads of course). Taking some thread, say thread j we would have the following:

(i) Thread j in its first pass will calculate relative to c_n using a value of $x = j$ as per the algorithm in figure 5.

(ii) Thread j will then store privately the value computed and from its point of view it will view this as the current value of the sequence.

[8] These approaches may have actually yielded acceptable random number generators, but they would not have been bit for bit RANMAR generators so they were not pursued.

(iii) Thread j will calculate its next sequence element relative to its own "current value" but using a value of $x = 32$ (that is, every thread is computing every 32nd value of the overall sequence)

This distinction between the first and subsequent passes constitutes an important optimisation, because without it, each thread would have to calculate relative to a common sequence value. This would mean having to synchronise all threads after each had computed their value and then having the last thread update some shared memory location with the "furthest" last value computed. Apart from concerns over the synchronisation aspect, the need to write to (and read from) shared memory means a performance penalty (over using purely thread local registers) and only having a single thread do this requires thread divergence (threads taking different code paths) which in GPU programming is generally highly undesirable, again for performance reasons.

Instead, with the described algorithm, once started, each thread can keep generating sequence values without synchronising or sharing data with other threads. While this approach describes an embarrassingly parallel computation, the arithmetic sequence cannot be considered in isolation. It must be combined with the LFG component of the RANMAR and therefore synchronisation points are still required. However, the optimisation is still valid because of the savings in shared memory access.

4.1.2 RANMAR Parallelisation Using Multiple Independent Sequences

The second method of parallelising the RANMAR is to have several independent random number sequences generated simultaneously and to have their results gathered into a single resultant sequence. The RANMAR, as pointed out by James in [13] lends itself to this because it is extremely easy to generate large numbers of independently disjoint sequences. In particular and as mentioned earlier, there are approximately 942 million sequences available with an average period of 10^{30}.

There are no particular challenges in implementing such an approach apart from two implementation considerations. Firstly, a decision was required on the method to be used for combining the independent sequences into the single output sequence. One element influencing this decision was the ultimate statistical quality of the combined sequence. Coddington [26] points out that combining multiple high-quality random number sequences into a single sequence will produce another high-quality random number sequence as long as the sub-sequences are seeded correctly and do not overlap (thereby introducing possible correlations). Given the fact that RANMAR's strength is in easily generating disjoint sub-sequences this issue was less of a concern (although as discussed later, the theory was validated using statistical software packages for testing random number sequence quality). The combination strategy ultimately chosen was: in a scenario where N random numbers are required from P processors, each processor P_i would generate N/P random numbers from its own sequence and assuming N is divisible by P, would populate an output random number array X as follows:

$$P_i \rightarrow X_{i.\frac{N}{P}} \ldots X_{(i+1)\frac{N}{P}-1} \quad \text{where } i = \{0..P-1\}$$

Or, by example: in a system where 10 processors are required to generate 100 random numbers, each processor generates 10 numbers from its own sequence. Processor 1 populates the first 10 numbers in the output array - processor 2 populates the second 10 and so on – processor 3 populates the third 10 and so on.

The second implementation consideration was around the seeding mechanism. In order to generate independent sequences for each processor, seeds are required for each one. It seemed unwieldy to require users to provide all these seeds - therefore a design decision was taken to keep the existing interface (that is, as per the RANMAR implementation in Corsika) and only require that the user provides 2 seeds - regardless of the number of independent sequences they require. The approach then taken was that the second user provided seed would be incremented by 1 internally (wrapping as necessary) before initialising each independent sequence.

4.2 Comparison of Design with Existing Schemes

As already discussed, an existing ATI graphics card implementation (Demchik in [14] and [23]) was discovered during the research phase of this project so it is perhaps instructive to compare his design with the current design. Quite apart from the fact that the design presented here is based on the CUDA (NVIDIA) architecture, there are other design differences that warrant further discussion. The primary difference is that the application of the "leapfrog" parallelisation strategy is unique to this implementation. Demchik uses the multiple independent sequences approach alone, but because of the nature of his problem, this was reasonable. His random number generator was part of a larger GPU computation where each GPU thread was allocated its own RANMAR sequence. Because of this, other differences arise between the two designs – for example, Demchik had no need to store the generated random numbers on the GPU and neither was he required to transfer them to CPU RAM. In contrast, the goal of the project presented here is to produce a general purpose random number generator for use by a CPU based simulation. Therefore there was no way to avoid the transfer of random numbers back to the CPU.

The leapfrog method was deemed appropriate for this project for the following reasons;

1. Because the GPU based RANMAR implementation is intended to replace an existing sequential implementation, there are likely repeatability requirements whereby the GPU RANMAR will be expected to generate exactly the same random number sequence as the sequential version. This is only possible when using a single RANMAR sequence. A single GPU thread generating that sequence would be prohibitively slow because it would have the normal RANMAR overhead along with the GPU call and data transfer overhead also. Additionally, GPU clock-speeds are generally substantially lower than those of CPUs. A single GPU thread would also be very wasteful, when one considers the processing power generally available on a GPU. On the other hand, a leapfrog approach to the RANMAR allows 32 GPU threads to cooperate in the generation of a single RANMAR sequence which at least means better use of the GPU computational capacity.

2. There is a larger memory footprint required of the purely independent sequences approach. This is because each sequence (which equates to each thread) needs to maintain the LFG array of 97 (presumably double precision) floating-point numbers as well as the current LFG index and the current value of the arithmetic sequence. A group of 32 threads performing the leapfrog method means that for a fixed number of threads, the overall memory footprint is reduced by a factor of 32.

4.3 Implementation Phase

The design phase was exclusively concerned with understanding the RANMAR itself and devising acceptable strategies for parallelising the algorithm. The implementation phase concerned itself with the details. The following sections describe the features and constraints required of the implementation. The various implementation challenges are then presented.

4.3.1 Corsika Imposed Constraints and Features

Mindful of the fact that the implementation would ultimately be integrated with the Corsika package, this imposed some particular requirements, including:

- Quite aside from the independent (sub)sequences approach to parallelising a RANMAR sequence, Corsika requires that it can initialise several independent sequences of the RANMAR and be able to later draw random numbers from a particular sequence of interest. All sequences are maintained internally by the RANMAR implementation and the caller only refers to them by id.

- During the initialisation phase, apart from providing the RANMAR seeds, Corsika requires that it can specify a particular number of random numbers to skip – this is to allow simulations to be started at any point in a given random number sequence.

- James [13] introduced the interface that allows RANMAR clients to request random numbers in batches (by providing an array and an array size argument). Corsika also requires this.

- There is an overhead associated with calling the GPU kernels, generating the random numbers and copying the results back to CPU memory. In order to absorb this overhead, each trip to the GPU should generate a substantial number of random numbers (at least 1 million). Unfortunately, an analysis of Corsika during simulations showed that it only requests random numbers in very small batches (~10). For this reason an interface was required that allows RANMAR clients to specify, during initialisation, a pre-fetch size, which constitutes the number of random numbers generated with each call to the GPU. The RANMAR caches the numbers in CPU memory and client requests are serviced from there, until the cache is exhausted and another GPU kernel call is required.

- The original RANMAR algorithm generates floating point random numbers with a 24 bit mantissa (single precision). Corsika requires a 48 bit (double precision) mantissa. Fortunately within the last two years, commodity NVIDIA CUDA enabled GPUs have introduced support for double precision floating-point arithmetic - otherwise this requirement may have presented a problem. There is a cost to pay over and above single precision performance, so the parallel RANMAR implementation presented here, provides a compilation option to generate random numbers with the preferred precision.

- The RANMAR specification generates uniform random numbers over the range [0, 1) however Corsika does not want any zero random numbers. A modification to the original RANMAR algorithm was therefore required which checked for a generated zero and replacing it with the smallest possible number according to the configured precision (that is, 2.0E-24 or 2.0E-48)

4.3.2 Other High Level Features

From a feature point of view, the Corsika requirements were the primary drivers. However, some more general requirements were also identified.

- The RANMAR implementation should not be coupled to Corsika (or any other client) in such a way that assumptions are made about how the RANMAR is used.

- Related to the previous point, the implementation should be shipped as a shared library with a well defined public interface.

- Two interfaces for generating random numbers are required. One that utilises the pre-fetching mechanism already described and a second where all requests for random numbers are serviced by a GPU kernel call – that is, without a pre-fetching mechanism.

4.3.3 Implementation Details and Challenges

The implementation of the GPU RANMAR used the sequential algorithm as its starting point and by considering the parallelisation strategies already discussed, the basis of the CUDA implementation was formed. Some early implementation decisions were made and they, along with the various algorithms used, are described in the following sections

4.3.3.1 Early Decisions

Because it is a sequential task, the initialisation of RANMAR state (that is, the initial LFG buffer population and setting the starting value for the arithmetic sequence) is performed on the CPU and copied to GPU global memory. Because GPU global memory persists across kernel calls, this state can remain on the GPU until the RANMAR instance is destroyed. There is little else to be said about the initialisation of a RANMAR instance, because once the seeds are provided, it is performed exactly as described in James [13] (apart of course from this being a C implementation).

In the discussion about parallelising a single RANMAR sequence it was noted that, in theory, 33 random numbers could be calculated simultaneously. However scheduling 33 threads on a CUDA GPU is not optimal because the warp size (that is the number of threads that are created, managed and scheduled by the GPU hardware as a single entity) is 32. A CUDA program that requested 33 threads in a thread block would make very poor use of the available processing power in the GPU because the first 32 threads making up the first warp would run simultaneously as desired, but the second warp, separately scheduled would only contain 1 thread and while it would reserve the same amount of hardware as the 32 thread warp, it would still only have 1 thread using it. For this reason, the RANMAR implementation hard-codes the use of 32 threads per RANMAR instance.

The multiple independent sequences strategy already discussed would be realised using an algorithm to allocate the generation of each sequence among CUDA thread blocks. The simplest model would be to allocate a separate thread block for each sequence computation, but given that each sequence computation uses 32 threads, there would only ever be 32 threads per thread block. This would not make optimal use of available GPU resources. Instead an algorithm would be employed to decide how many sequences should be allocated to each CUDA thread block.

4.3.3.2 High Level Algorithm

The implementation was then required to bring all the individual requirements, strategies and decisions into a unifying algorithm. Putting aside for a moment the aforementioned approach of caching random numbers generated by the GPU in CPU memory and assuming also that the RANMAR state has already been initialised on the GPU, the high level algorithm for retrieving a specified number of random numbers from the GPU is as follows:

- From the CPU, an array (of a size to hold the requested number of random numbers) is allocated in GPU global memory using the standard CUDA memory allocation API.

- Again from the CPU, two arrays (each sized according to the number of RANMAR instances) are allocated in GPU global memory. These arrays will be used later from the GPU kernel when each RANMAR instance will get a start index from one array and an end index from the other array, telling it the range of random numbers for which it is responsible.

- Based on the number of instances (independent sequences) and on the number of required random numbers, the random number array is divided up (as evenly as possible) among the RANMAR instances by populating the start and end index arrays.

- The RANMAR GPU kernel is launched with the specified necessary number of thread blocks, (based on the algorithm referred to in section 4.3.3.1), and with each having the some multiple of 32 threads (that is 32 threads for each sequence computation assigned to each thread block).

- Finally, the random numbers in the random number array on the GPU, having been populated by the kernel invocation are copied back from the GPU into the CPU memory array provided by the RANMAR client. The GPU memory that was allocated is also freed at this time. GPUs tend to be limited in memory compared to CPUs so a memory leak, even a small one, can be quickly catastrophic.

Figure 6 illustrates the algorithm more clearly by way of an example. Here the RANMAR has been initialised with three instances (sequences), so during initialisation 3 LFG buffers would have been allocated in GPU global memory and populated according to the RANMAR initialisation procedure. At some later point, 300 random numbers are requested, so the algorithm allocates a 300 element array in GPU memory and will assign index ranges in this array to the three instances (in this case, it is evenly divisible so each instance gets a range of 100) by allocating and populating the indices arrays shown. When the kernel is launched, each GPU thread works out the sequence of which it is a part (by referencing to the usual CUDA grid and block dimension variables). It then reads its own start and end indices and using its own LFG state (and arithmetic sequence state which is not shown) generates its assigned random numbers. The large arrow indicates the copying of the entire random number array to CPU RAM when the kernel returns control to the CPU.

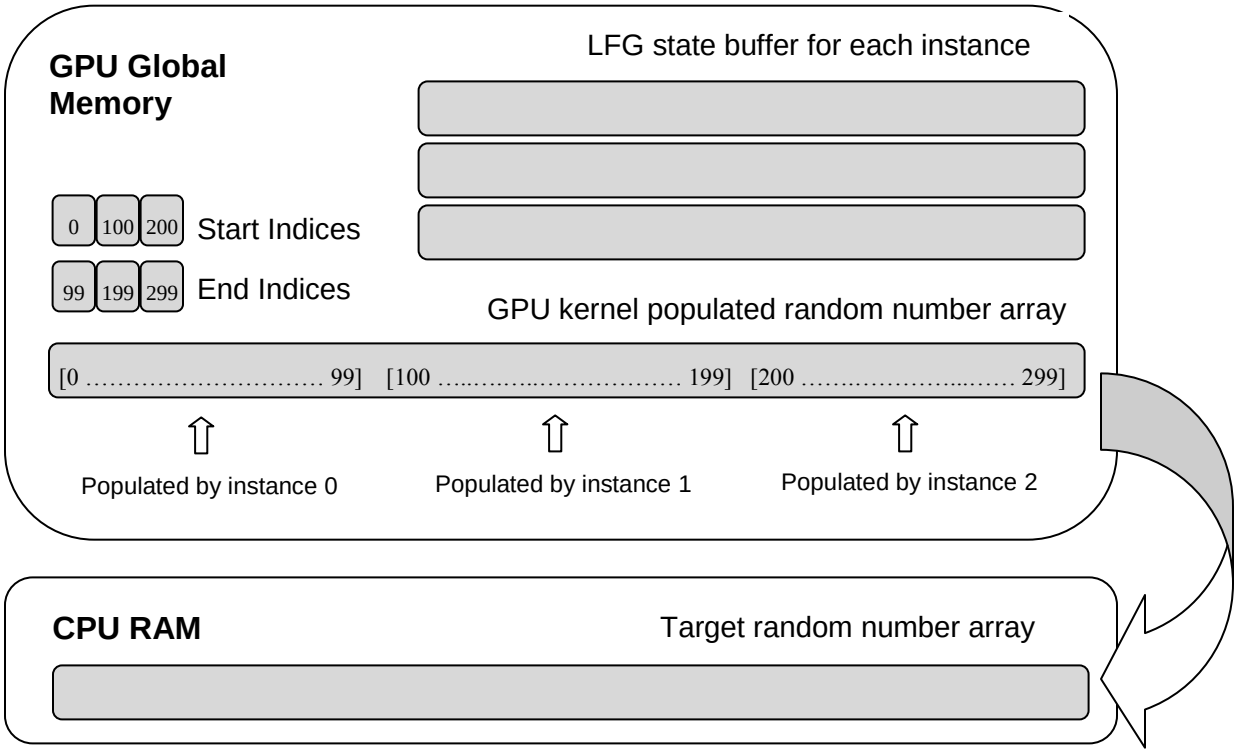


Figure 6. Example memory usage when RANMAR has been initialised with 3 instances and later 300 random numbers are requested. The start and end indices buffers inform each sequence which areas of the random number buffer they are to populate.

4.3.3.3 Kernel Algorithm

The final aspect of the RANMAR implementation is the GPU kernel itself. Although much of what happens here, such as the key parallelisation approaches, has already been explained, many details remain. Given that the kernel execution (and its performance) is so central to the project's goals, it is an area that deserves some detailed explanation. Perhaps the most appropriate place to start is with an overview of the primary performance considerations that informed the RANMAR kernel (or indeed any kernel) implementation.

One the most important performance considerations is how a kernel accesses memory or more particularly - global memory. Global memory access, especially on early generation GPUs without a cache hierarchy, is extremely slow, so any strategies for minimising global memory use (even on next generation GPUs) is generally recommended. The RANMAR kernel for example performs all its global memory reads upon kernel start-up and copies performance critical data to shared memory – for example, the LFG state array. Any data that does not need to be shared across threads is copied into thread local registers – for example the arithmetic sequence state. Unfortunately, given that the purpose of RANMAR is to generate random numbers, there is nothing that can be done about the writes to global memory. On the plus side however, the RANMAR helps with memory store coalescing (making concurrent memory accesses with a single load or store operation across multiple threads) because contiguous threads store their random numbers to contiguous global memory addresses.

While RANMAR was primarily concerned with global memory, it was possible to move all critical data to thread registers and shared memory. Other situations may require more sophisticated solutions, for example, using other areas of memory, such as constant or texture memory. Sometimes algorithms may need to be completely redesigned so as to have a more performance friendly memory access profile.

Another critical factor in kernel code is to minimise thread divergence. This happens when "if-then-else" (or equivalent) constructs causes some threads to take different code paths. Any threads not taking a particular path become idle and must wait for the other threads to finish their execution path - then there is another code pass required for the threads that take the other code path. In effect the execution moves from parallel threads to serialised threads and arises because the thread warp (32 threads) is scheduled as a unit and so all threads within the warp must execute the same instruction. While it is often impossible to completely remove thread divergence, minimising it is critical.

Some operations are particularly expensive such as floating point division and integer modulus. Some experimentation during the RANMAR testing discovered the large impact a modulus operation could have on overall performance when comparing two options for implementing wrapping around an incrementing index in a circular buffer.

```
// Option 1: avoids thread divergence so might be preferred
r = (r+32) % BUFFER_LENGTH

// Option 2: despite the thread divergence, entire RANMAR
```

```
// runs 4% quicker this way
r+=32;
if (r >= BUFFER_LENGTH)
      r = r - BUFFER_LENGTH;
```

Making full use of the available computational power of the GPU is obviously important. In the RANMAR case we are constrained somewhat because the nature of the random number generator is such that only 33 numbers can be computed simultaneously within a single RANMAR instance. With mid level GPUs now having up to 500 CUDA cores, a single RANMAR instance with only 32 threads would be making very poor use of the available horsepower. The solution in the RANMAR case was to use the multiple independent sequences approach and to assign more than one such sequence to the same CUDA thread block. Multiple thread blocks were also used. With each sequence being calculated by 32 threads and with say 5 such calculations per thread block and then with 10 such thread blocks, there would be: 32 x 5 x 10=1600 threads.

The CUDA C Programming guide [25] and the NVIDIA published CUDA programming books [1] and [2] were invaluable resources in determining the best strategies for maximising kernel performance. The kernel source code delivered with this thesis is heavily commented and describes many of the performance aspects in specific detail, but for the sake of brevity, only a high level algorithm is described here.

- The kernel is launched from the CPU with a one dimensional grid of thread blocks. Each thread block has a two dimensional thread structure. The first dimension represents a grouping of the threads working on the same RANMAR instance. The second thread dimension is the hard-coded size 32 grouping of threads.

- Inside the kernel thread, it first establishes which RANMAR instance it belongs to. It calculates this by multiplying the block dimension (how many instances per block) by the block index and adding the first dimension of the thread dimension structure. Note that the second thread dimension indicates which thread (0 to 31) is executing.

  ```
  instanceId = (blockIdx.x * blockDim.x) + threadIdx.x
  ```

  If the instance id is greater than the total number of instances then the thread exits immediately. This can happen because there are a fixed number of instances allowed per thread block, but the end user may request any number of instances which will probably not be an exact multiple which ultimately means some threads have nothing to do.

- Each kernel thread calculates the index into the output random number array that it will populate based on the start index and end index arrays and on the thread id. Once the number generation starts, this thread specific index will be incremented by 32 (the leap-frog) for each iteration.

- One of the threads for each RANMAR instance copies the LFG buffer for that instance into shared memory for performance reasons. Threads are synchronised at this point.

- Each kernel thread calculates the index into the LFG buffer that it will write to next and based on that will also work out the initial lag indices for the LFG calculation.

- Each thread then works out how many random numbers it needs to produce (not all threads will produce the same number).

- Next the RANMAR loop begins. The logic is similar to the sequential algorithm except all indexes are incrementing by 32 instead of 1. Also the arithmetic sequence implementation is as per the algorithm described in Figure 5. For each number generated, it is copied into the global random number array. In this manner all threads from all sequences are populating different regions of this shared array.

- At loop exit, one thread for each RANMAR sequence copies the LFG state and arithmetic sequence back to global memory so that the next invocation of the kernel can restart from the same place in each sequence.

4.4 Verification of RANMAR Correctness

The first task post implementation was to verify that the algorithm and the corresponding random numbers produced were correct. This was done by first implementing a sequential RANMAR and verifying its correctness by reference to Marsaglia et al in [12]. For a particular seeded RANMAR sequence, we are told the values of the random numbers from positions 20,001 to 20,006 in the sequence. A simple comparison is sufficient to have confidence in the sequential algorithm.

Next, the sequential algorithm was used to verify the values for a number of different RANMAR sequences generated by the GPU implementation. The first 100 billion numbers in each sequence were checked and this it was felt was more than sufficient to have confidence in the correctness of the algorithm. By definition, this verification could only be performed on the GPU RANMAR when it used a single RANMAR sequence. For multiple RANMAR sequences combined into a single sequence, it is the quality, or randomness of the numbers produced that must be verified and this is discussed separately in section 5.

4.5 Integration with Corsika

The final implementation task required the integration of the GPU RANMAR with Corsika. This had its own challenges because it is written in FORTRAN an area with which the author was unfamiliar. However, all that was really required was to replace the implementation of the internal RANMAR implementation with a call to the GPU version contained in a shared library. Fortunately, calling C functions from FORTRAN

subroutines is very straightforward, apart from a couple of quirks (such as passing arguments by pointer only).

Interestingly, the Corsika build environment runs the C pre-processor on its FORTRAN source code to allow the familiar #ifdef, #else, #endif paradigm. This allowed the changes made to call GPU code instead the Corsika RANMAR easy to swap in and out during development. One slight annoyance however was the fact that rather than there being one random number generator subroutine to change, there were in fact a dozen, all almost identical and with the kind of differences that could easily have been parameterised. This multiplicity of implementations made it harder to consider making larger changes to the Corsika code base beyond making the call out to the GPU RANMAR.

5 Statistical Validation of Generator Output

5.1 Validation Approach

Statistical analysis of the output of the parallel RANMAR was deemed necessary due to the decision to use multiple independent sequences and to combine their output into a single sequence. While it may seem intuitive that multiple random number sequences spliced together in this way should yield another statistically valid random number sequence, this is not necessarily so (as mentioned in [26]). The combination process, perhaps due to poor seed choices, could yield correlations not present in the original sequences. For this reason it was decided to apply thorough statistical testing using TestU01 [27] - the respected random number generator tester. TestU01 supports the easy application of predefined batteries of statistical tests to random number generators. The available test batteries are known as: small crush, crush and big crush (in order of increasing strictness).

TestUO1 requires that a test program be written to enable the tester to draw numbers directly from the random number generator under test. The alternative approach, as used in some earlier testers such as Diehard (also by Marsaglia) is to draw its sample of random numbers from a user provided file. However, this file based approach is impractical for the more strict randomness tests, because these tests typically require a very large quantity of random numbers. The programmatic approach on the other hand, once initially configured, can cater for the most thorough of statistical tests and the code required is not onerous. Some sample code is presented in figure 7.

Before discussing the tests that were performed with TestU01, it is probably worth mentioning that another popular random number generator tester, Dieharder (the tongue in cheek successor to Diehard) was also tried. Unfortunately, attempts to get any tests to pass, even when run on Dieharder's own RANMAR implementation failed. Time did not permit any investigation into the cause of the problems and Dieharder was abandoned in favour of TestU01.

For an exhaustive description of the actual statistical tests performed by TestU01, see [27]. The high level test plan using TestU01 was as follows:

- Perform a sanity validation on the double precision sequential RANMAR provided by TestU01.
- Perform a sanity validation of the double precision parallel RANMAR initialised with 1 instance (which as verified previously, generates the same numbers as an equally seeded sequential RANMAR)
- Perform a validation of the double precision parallel RANMAR initialised with 8 instances.

```
double get_double(){
    static ranmar_t r;
    return (double)ranmar_get(0, &r, 1);
}
int main(int argc, char* argv[]) {
    /* Initialise RANMAR with 4 instances and prefetch 100000 */
    int ij = 1802; int kl = 9373;
    ranmar_initialise(0, ij, kl, 4, 100000);

    /* Run big crush on parallel RANMAR with 1 instance */
    unif01_Gen* gen = unif01_CreateExternGen01(
          "RANMAR", get_double);
    bbattery_BigCrush(gen);
}
```

Figure 7. Driver program to run big crush on a RANMAR instance

5.2 Validation Results

5.2.1 Sanity Validation of Sequential RANMAR Provided by TestU01

In an attempt to set a baseline expectation on the statistical quality of the RANMAR according to TestU01, the TestU01 internal RANMAR implementation was tested. The initial results were surprisingly poor, even on the small crush (most forgiving) battery. Some investigation uncovered a problem in the TestU01 RANMAR implementation whereby the double precision RANMAR was really only generating single precision values (initialising only 24 bits). With half of the mantissa bits always zero, unsurprisingly, several tests failed. The problem was rectified with a small code change and TestU01 itself was recompiled. The test was then run again and the results of the crush battery were more promising:

```
========= Summary results of Crush =========
Version:          TestU01 1.2.3
Generator:        Sequential RANMAR
Number of statistics:  144
```

```
The following tests gave p-values outside [0.001, 0.9990]:

    Test                            p-value
------------------------------------------------
22  ClosePairsBitMatch, t = 4       2.2e-4
34  Gap, r = 22                     5.3e-6
54  WeightDistrib, r = 24            eps
------------------------------------------------
All other tests were passed
```

5.2.2 Sanity Validation of Parallel RANMAR with 1 Instance

The sanity check on the single instance parallel RANMAR was, as expected, identical to the sequential result.

```
========= Summary results of Crush =========
Version:          TestU01 1.2.3
Generator:        Parallel RANMAR 1 instance
Number of statistics:  144
The following tests gave p-values outside [0.001, 0.9990]:

    Test                            p-value
------------------------------------------------
22  ClosePairsBitMatch, t = 4       2.2e-4
34  Gap, r = 22                     5.3e-6
54  WeightDistrib, r = 24            eps
------------------------------------------------
All other tests were passed
```

5.2.3 Validation of Parallel RANMAR with 8 instances

The first test of the parallel RANMAR with multiple instances was performed using 8 instances and run 10 times with different seed values each time (each run being the crush test battery and taking a little over an hour and a half). Each test run yielded the same test failure profile. Interestingly, this test failure profile was actually better than the sequential RANMAR because it passed the ClosePairsBitMatch test. This was an important result because it validated the decision to parallelise the RANMAR using multiple independent sequences (instances), and it also validated the method of seeding the RANMAR instances (by incrementing the second of the two user provided seeds).

```
========= Summary results of Crush =========
Version:          TestU01 1.2.3
Generator:        Parallel RANMAR 8 instances
Number of statistics:  144
The following tests gave p-values outside [0.001, 0.9990]:

    Test                            p-value
------------------------------------------------
34  Gap, r = 22                     6.4e-5
54  WeightDistrib, r = 24            eps
------------------------------------------------
```

```
All other tests were passed
```

5.3 Proposed Extension to Current Double Precision RANMAR

While the validation of the independent sequences approach was welcome, there was a suspicion that improvements could be made. The RANMAR as originally described was a single precision random number generator – that is, it generated 24 bit random numbers by combining 24 bit elements from an LFG sequence with 24 bit elements from an arithmetic sequence. As mentioned in section 2.4.1, when RANMAR was extended to double precision in the CERN program library, the only change made was to extend the initial values of the LFG buffer to 48 bits (thereby making all subsequent LFG values 48 bit also). The initial value of the arithmetic sequence, its decrement value and its modulus remains 24 bit. This means that the process of combining the 48 bit LFG with the 24 bit arithmetic sequence (a subtractive and additive operation) causes only the high order 24 bits of the LFG values to change. The low order 24 bits remain unaffected. This means that the low order 24 bits of the RANMAR sequence are derived purely from the LFG. Given the concerns expressed by Marsaglia et al about the randomness of the LFG on its own, there is valid reason to be concerned about the double precision RANMAR implementations present in the CERN program library and in Corsika.

In order to test this hypothesis, an alternative arithmetic sequence was implemented, one where the parameters defining the arithmetic sequence were converted from 24 bit fractions to 48 bit fractions. The GPU RANMAR implementation used 64 bit longs in its computations before converting the final arithmetic sequence value to a 48 bit fraction. The following sections describe the TestU01 results of the new implementation.

5.3.1 Validation of Extended Parallel RANMAR with 1 instance

This test of the new extended RANMAR implementation used the TestU01 big crush battery as it was necessary to give any changes to the generally accepted form of RANMAR, the sternest possible examination. Once again a comparison baseline test was performed, using the standard (non-extended) parallel RANMAR.

```
========= Summary results of BigCrush =========
Version:   TestU01 1.2.3
Generator: Double precision parallel RANMAR 1 instance
Number of statistics:  160
The following tests gave p-values outside [0.001, 0.9990]:

    Test                          p-value
-----------------------------------------------
35  Gap, r = 25                     eps
61  WeightDistrib, r = 28           eps
64  WeightDistrib, r = 26           eps
98  HammingIndep, L=300, r=26      1.2e-12
-----------------------------------------------

All other tests were passed
```

The next test was on the newly extended double precision RANMAR, again using the big crush battery. The results here were generally very encouraging (although only two such tests were possible because of the time taken to run - up to 18 hours). In summary, it was found that the extended RANMAR passes more tests than the standard double precision RANMAR found in the CERN program library and in Corsika. More investigation would be required to establish if this trend is repeated across many different RANMAR sequences, but time did not permit this. It is worth noting however, that the testing of random number generators is in itself a prime candidate for parallelising and implementing on a GPU.

```
========= Summary results of BigCrush =========
Version:   TestU01 1.2.3
Generator: Double precision extended parallel RANMAR 1 instance
Number of statistics:  160
The following tests gave p-values outside [0.001, 0.9990]:

    Test                              p-value
-----------------------------------------------
60  WeightDistrib, r = 20              9.5e-6
61  WeightDistrib, r = 28              1.2e-5
-----------------------------------------------
All other tests were passed
```

5.3.2 Validation of Extended Parallel RANMAR with 8 instances

The final test executed the big crush test battery against the multiple instances version of the extended RANMAR. Again, a baseline test against the non-extended parallel RANMAR was performed first. The results were almost identical to the single instance parallel RANMAR.

```
========= Summary results of BigCrush =========
Version:   TestU01 1.2.3
Generator: Double precision parallel RANMAR 8 instances
Number of statistics:  160
The following tests gave p-values outside [0.001, 0.9990]:

    Test                             p-value
-----------------------------------------------
35  Gap, r = 25                        eps
61  WeightDistrib, r = 28              eps
64  WeightDistrib, r = 26              eps
98  HammingIndep, L=300, r=26        5.2e-14
-----------------------------------------------
All other tests were passed
```

The results of the extended RANMAR with multiple instances, again shows an improvement over the non-extended version. The extended version in the tests performed passed the Gap and Hamming independence tests that were observed as failing in the non extended RANMAR test.

```
========= Summary results of BigCrush =========
Version:   TestU01 1.2.3
Generator: Double precision extended parallel RANMAR 8 instances
Number of statistics:  160
The following tests gave p-values outside [0.001, 0.9990]:

    Test                          p-value
------------------------------------------------
60  WeightDistrib, r = 20          3.2e-4
61  WeightDistrib, r = 28          4.8e-6
------------------------------------------------
All other tests were passed
```

6 Performance Results

6.1 Introduction

The RANMAR implementation was tested on two systems (the details of which are presented in Appendix A). The first system was a desktop personal computer with an Intel Pentium 4 CPU (at 3.2GHz) with the GPU being an NVIDIA GeForce GTX 460. The second system was a modern server machine with a Xeon X5670 CPU (at 2.93GHz) with the GPU being an NVIDIA Tesla C2050. Before any tests were run, it was expected that the server machine would outperform the personal computer because of the Xeon being a more modern CPU and because the personal computer only had a PCI express 1.0 bus (for graphics card connectivity). The transfer of random numbers from the GPU to CPU can dominate the runtime and PCI express 2.0 offers a potential doubling of throughput when compared with version 1.0. Despite the differences, it was deemed instructive to test the RANMAR on more than one system.

6.2 Standalone Testing

The first and most important performance testing was standalone testing as this would provide the data that could predict the kind of speedup possible with RANMAR integrated with Corsika. These tests involved the use of a client program to drive the random number generator in various scenarios in order to characterise the performance of the GPU parallel RANMAR and also to compare it with the sequential RANMAR. In each scenario, the RANMAR is asked to generate 1 billion random numbers. The scenarios tested were as follows:

1. For the non-buffered API (each client request involves a call to the GPU), the quantity of random numbers requested on each trip to the GPU is varied.

2. For the buffered API (where numbers are cached by the RANMAR in CPU RAM), the size of the pre-fetch is varied.

3. For each of the first two scenarios the number of RANMAR instances is varied.

4. For the first 3 scenarios, the scheme of allocating instances to CUDA thread blocks is varied – either 1 instance per thread block, or 4 instances per thread block.

There are some other characteristics of the standalone tests that should be made clear before presenting results:

- In the sequential tests on both systems, each individual call to the sequential RANMAR requested 10 random numbers (therefore 100 million calls were necessary to generate the required 1 billion numbers). The same buffer was reused for each call so there is no repeated memory allocation overhead. The number 10 was chosen based on profiling Corsika's use of the RANMAR and observing that the average RANMAR call requested 10 numbers.

- For the reason outlined in the previous point, in the buffered parallel tests on both systems, each individual call to the parallel RANMAR requested 10 random numbers. This was practical because the requests could usually be serviced from the CPU RAM cache without needing to visit the GPU.

- In the non-buffered parallel tests, because each call to RANMAR requires a call to the GPU, the tests requested far larger batches of random numbers.

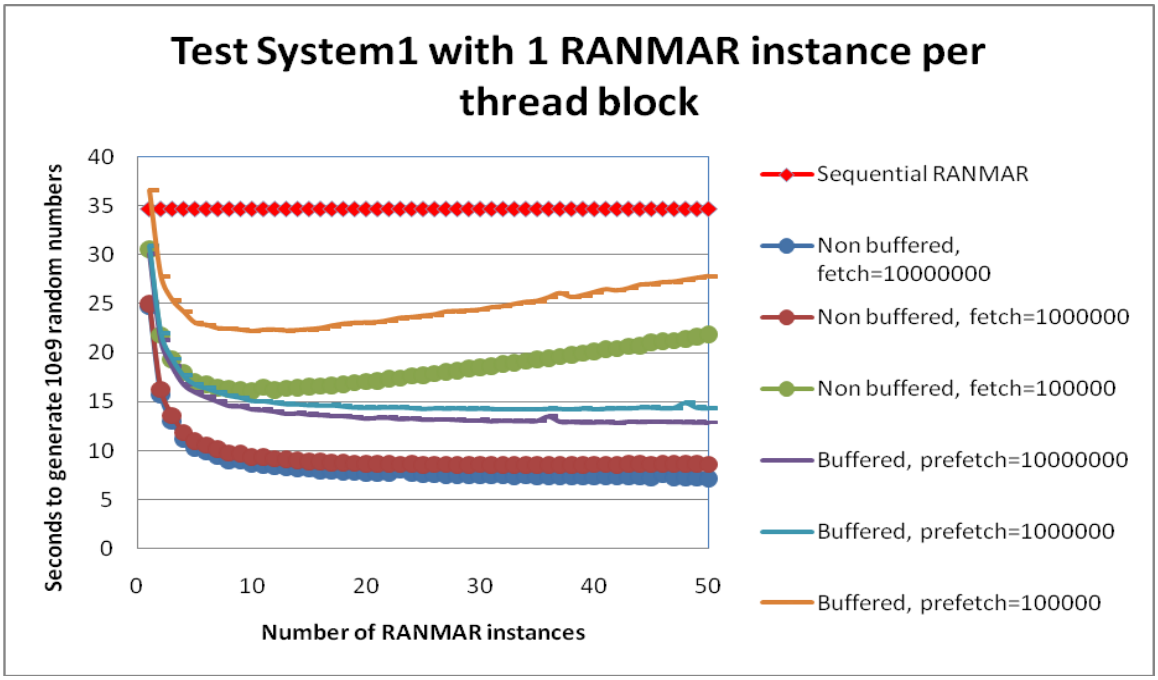


Figure 8. Varying RANMAR instances on PC system with 1 instance per thread block

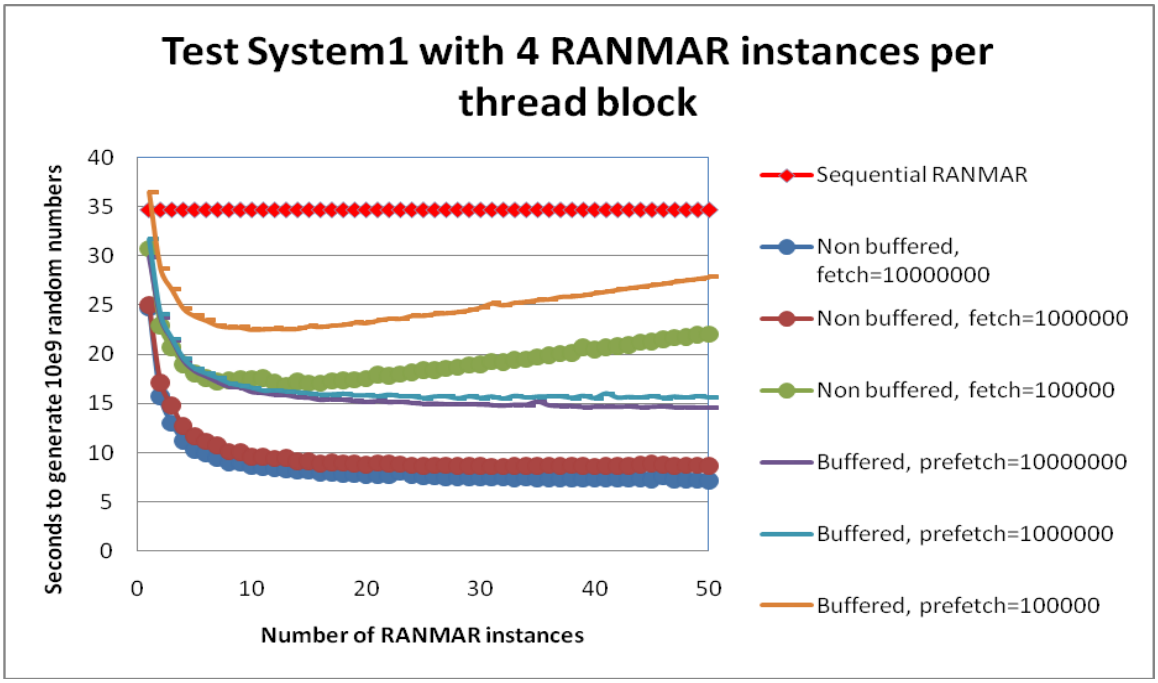


Figure 9. Varying RANMAR instances on PC system with 4 instance per thread block

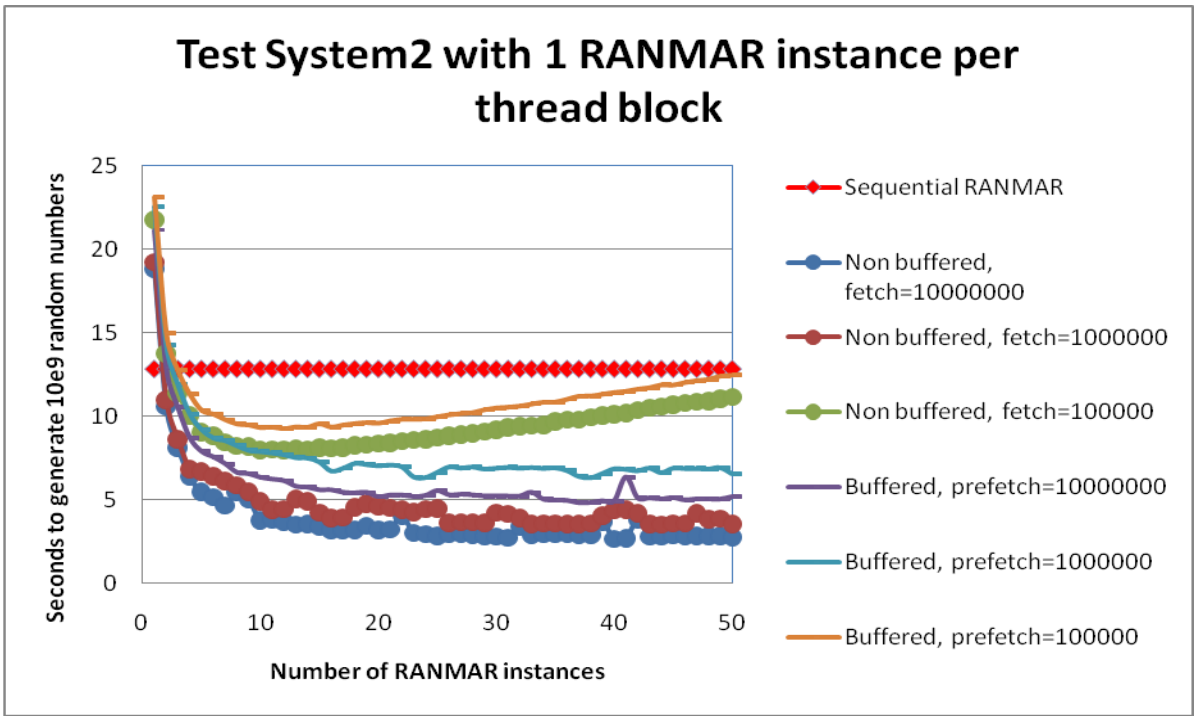


Figure 10. Varying RANMAR instances on server system with 1 instance per thread block

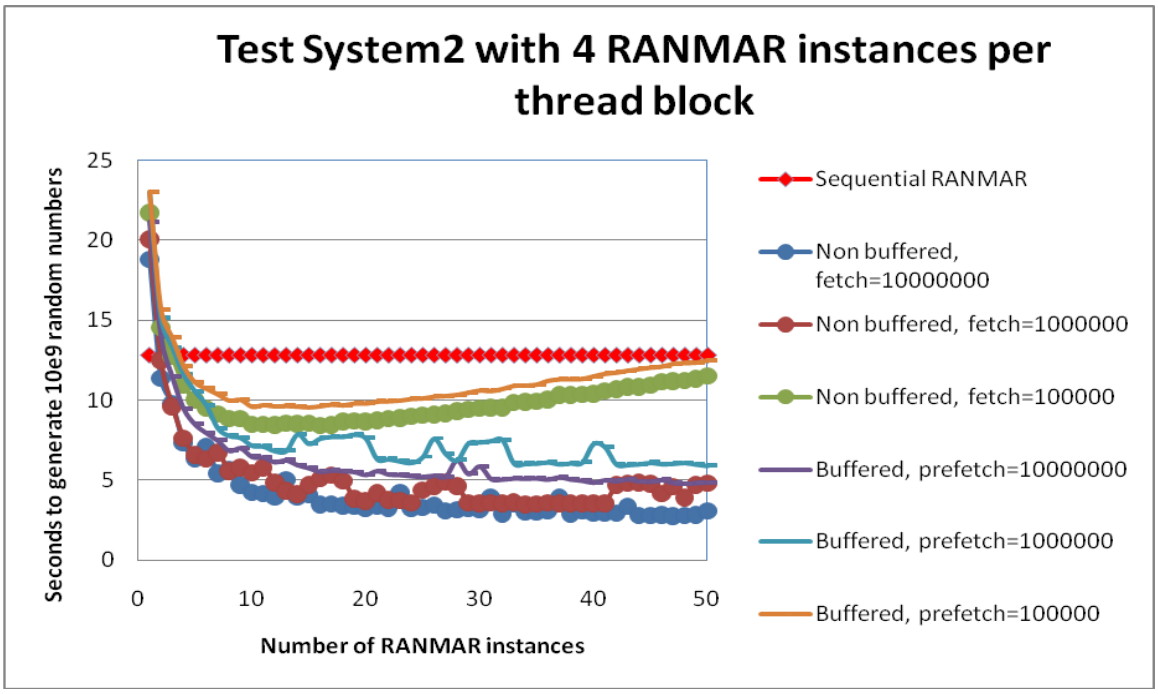


Figure 11. Varying RANMAR instances on server system with 4 instances per thread block

6.2.1 Analysis

The results in figures 8 to 11 enable a number of conclusions to be drawn:

1. The straight line sequential (CPU) speed of the server system is greatly superior to the PC system. The indicative time to generate 1 billion numbers on the PC was 34.65 seconds whereas on the server system it was 12.83 seconds.

2. The best observed RANMAR performance result on the PC system was:

    - Best non-buffered speedup over sequential: 4.82.
    - Best buffered speedup over sequential: 2.69
    - Best scenario: non-buffered, fetch size=10,000,000.

3. The best observed RANMAR performance result on the server system was:

    - Best non-buffered speedup over sequential: 4.85
    - Best buffered speedup over sequential: 2.68
    - Best scenario: non-buffered, fetch size=10,000,000.

4. There was no definitive performance difference between allocating multiple instances to the same thread block versus having only 1 thread per block. This would seem to suggest that the performance of CUDA kernels does not change whether warps are scheduled as part of the same thread block or there is only one warp per thread block (at least within the bounds actually tested here).

5. As might be expected, the best GPU RANMAR performance is achieved using the non-buffered interface and when the fetch size is maximised. The reason the non-buffered interface outperforms the buffered counterpart is that the buffered interface has extra memory copying to perform – once from the GPU to the RANMAR cache and then again to a client provided array when the call to request random numbers is made. The non-buffered interface can copy random numbers directly into the client provided array.

6. The graphs show that in the best case scenario (on both systems) using the non-buffered interface with the largest fetch size, the performance quickly reaches a point of diminishing marginal returns as the number of instances is increased. An analysis of the kernel runtime using the CUDA profiler shown below in figure 12, in one particular test run provided the reason:

```
Total time for program execution   : 2.926 sec
Time copying data from GPU to CPU  : 2.165 sec
Time copying data from CPU to GPU  : 0.019 sec
Time spent executing kernel        : 0.512 sec
Time spent in all other processing : 0.230 sec
```

Figure 12. CUDA profile of best RANMAR scenario on the server test system

From this it is clear that the performance of the parallel RANMAR is dominated by the performance of the transfer of random numbers generated from GPU memory to CPU memory. This profile alone suggests that only 17% of execution time is spent generating the numbers whereas 74% is in transferring data from GPU to CPU. This suggests that at this point, the kernel implementation is close to optimal (in the current system context). Any further performance gains will be achieved through improvements in GPU/CPU transfer throughput.

7. Given that little performance gain is possible from maximising use of the GPU, there appears little point in initialising parallel RANMAR generators with large numbers of instances (say >25).

8. In the worst performing scenarios, where smaller numbers of random numbers are generated for each call to the GPU, adding extra instances means that each instance has even less to do and extra instances can actually cause overall performance to degrade. In figure 8, the buffered scenario with a pre-fetch of 100,000 spread over 50 instances means that each instance is only generating 2000 random numbers. GPU algorithms perform best when there is enough computational workload to absorb the cost of the CUDA overhead.

6.3 Corsika Testing

The second area for performance testing involved comparing Corsika simulations using the internally provided RANMAR implementation against simulations using the GPU based RANMAR. As already mentioned, Corsika generally requests random numbers in

quite small batches - averaging about 10 for the simulations profiled, but frequently it will ask for one random number at a time. This means that Corsika was forced to use the buffered interface to the GPU RANMAR. The results from the standalone performance testing in section 6.2.1 showed that the best expected speedup from the server test system using a buffered interface to the RANMAR would be approximately 2.68.

Given the time invested in integrating the GPU RANMAR with the Corsika system, not enough time remained to fully explore all the possible Corsika simulations. This was due to the fact that Corsika is quite large and it includes many possible interaction models for use in cosmic radiation simulations. Some analysis was of course possible but ultimately, the GPU RANMAR implementation will be presented to Corsika domain experts for further analysis.

With the performance of the GPU RANMAR implementation already characterised from the standalone testing, it only remained to find a Corsika simulation with the largest time spent generating random numbers. By so doing, the best possible overall simulation speedup would be realised. In the time available, by using the Gnu FORTRAN profiler, a simulation that spent 32% of its time in RANMAR was found. (Note: from initial discussions with the UCD School of Physics, it was said that there were simulations spending 80% of their time generating random numbers – but this has yet to be verified). The best results achieved are presented in figure 13.

```
Chosen simulation RANMAR % time          : 32%
RANMAR instances                         : 20
RANAR pre-fetch buffer size              : 10,000,000
Simulation speedup due to GPU RAMNMAR    : 17%
RANMAR speedup as part of simulation     : ~X2
```

Figure 13. Best overall Corsika simulation speedup observed.

The results of the Corsika performance profiling show that the overall simulation speedup achieved using the GPU RANMAR is broadly in line with that predicted by the standalone testing. While the 2.68 speedup was not observed, it was noted that the GPU RANMAR imposes an extra function call overhead on Corsika which may explain the disparity.

7 Conclusion and Future Work

7.1 Project Recap

In this project, an overview of the technology trends driving mainstream software engineering towards general purpose programming on GPUs was presented. The NVIDIA CUDA architecture was researched (primarily from [1] and [2]) and used to produce a novel and successful implementation of the well known RANMAR random number generator. The problem domain and the motivation behind the development of a GPU based RANMAR was explained through a description of the Monte Carlo

simulations that use it. The implementation was novel in that no implementation of the RANMAR could be found using the CUDA architecture on NVIDIA GPUs. Moreover, the implementation used two levels of parallelisation: multiple independent sequences and leapfrog parallelisation within each sequence. This two fold approach provided a more sophisticated implementation than any other RANMAR implementations that were found. The challenges involved in realising the parallelisation strategies, in particular the leapfrogging of the arithmetic sequence aspect of the RANMAR were detailed. The algorithms in the control program and the GPU kernel were also explained.

Given that the overriding goal of the project was to increase the speed of an existing CPU RANMAR implementation, several strategies were employed to maximise the performance of the GPU based implementation. Through program timing and CUDA profiling, it was shown that the best case tests of the implemented GPU RANMAR were constrained by the transfer of random numbers from GPU to CPU memory and not by the RANMAR algorithm itself. Almost a 5 fold speedup over a sequential RANMAR implementation was achieved during standalone testing. When integrating with Corsika, (with an extra function call random number copy overhead) the observed speedup was 2. This was the target set during the initial planning stages for this project, which from a performance point of view renders the project a success.

The statistical quality of the random numbers produced by the GPU RANMAR using the multiple independent sequences was verified using TestU01. Finally, statistical evidence of a possible weakness in the double precision RANMAR implementations found in the CERN program library and in Corsika was found. An improved double precision RANMAR was proposed and implemented, and shown to pass more tests from the most stringent of the test batteries from TestU01 (“big crush”).

7.2 Conclusions

The advertised strength of the GPUs is in computation on data parallel problems. For best possible performance there should be a high proportion of computational complexity relative to the amount of data to be transferred between the CPU and GPU memories (and the amount of GPU global memory accesses). For this reason, it was always of concern that random number generation may not have been a viable problem to implement on a GPU. The evidence in this project suggests that while the multiple orders of magnitude speed-ups may not be seen, a speed-up of up to 5 is still possible. The performance of the RANMAR was shown to be constrained by the transfer of the generated random numbers from GPU back to CPU, so this is the area in which further improvements will be made – possibly via a next generation bus (a successor to PCI Express) or more probably by a unification of the CPU and GPU such that the transfer isn’t in fact necessary. Hybrid CPU/GPU chips are already production, for example, AMD Fusion, Intel Sandy Bridge or NVIDIA’s Project Denver (which is in development). More generally however, these hybrid chips provide the most likely trajectory for data parallel SIMD style programming to enter software engineering’s mainstream.

The CUDA programming model proved very useful and does not distract the programmer from the actual problem domain – that is, it does not have a complicated API and neither

does it introduce any tricky syntax. Some aspects were of slight concern such as being able to extract significant performance gains by simply reordering some source code. This sort of trial and error programming is not something normally seen on CPUs due to mature and aggressive optimising compilers and CPU optimisation strategies such as multi-level caches, hyper-threading, branch prediction, instruction reordering, register reassignments and so on. It is likely that GPU compilers and chips will improve in the future however.

As would be expected with any parallel architecture, programming with CUDA requires a particular data parallel mindset. In addition, algorithm design using CUDA, on current generation GPUs requires the programmer to be very aware of the underlying hardware architecture. The relative performance of the various GPU memories and the constraints on aspects such as registers, shared memory size and maximum threads per thread block must always be considered. Sometimes tradeoffs are necessary – for example, moving computation onto the GPU despite it running slower there, in order to save on an expensive data transfer between GPU and CPU.

The RANMAR implementation was successful because it reached its target of a two fold speed up when integrated with Corsika. However, the RANMAR lent itself to parallelisation by: (a) having suitable LFG lags (b) using an arithmetic sequence and (c) it being easy to generate multiple disjoint sequences. Other random number generators may not have these advantages so not all generators will be parallelisable in the same way.

The suggested weakness in the double precision RANMAR in the CERN program library and in Corsika requires further investigation. This project makes a compelling logical argument as to why these implementations are questionable, and it produces some statistical evidence supporting this claim. The extended RANMAR implementation that addresses these concerns has been shown, in certain tests, to produce a statistically superior random number sequence. However a much larger series of tests would be required to confidently assert this.

7.3 Future Work

While a doubling of RANMAR performance in the Corsika context was welcome, a further increase in performance could be achieved by implementing the following:

- **Dedicated CPU thread for generating random numbers**: Assuming there are available CPU cores, it may make sense to have a dedicated CPU thread populating a CPU buffer of random numbers ensuring there are always numbers available when simulations need them. A single reader/single write thread queue mechanism such as this can be implemented very efficiently with so called lock free methods.

- **Change the way Corsika draws down its random numbers**. Corsika itself could maintain a single large array that it populates only as needed with calls to the GPU (effectively, the non-buffered approach except managed by Corsika itself). Corsika simulations that require a certain quantity of random numbers

simply request that quantity and are returned a pointer to the next random number in the array (assuming there are enough numbers left to satisfy the request, otherwise a GPU call is necessary). The result would mean a significant reduction in memory copying and a reduced function call overhead.

- **Move simulations onto the GPU**: If Corsika Monte Carlo simulations ran on the GPU then quite apart from the potential gains that that would bring, the RANMAR performance would also increase because the need to transfer the generated random numbers from GPU to CPU would disappear.

In addition, given that the future of general purpose GPU programming appears likely to trend towards the aforementioned hybrid CPU/GPU chips, a RANMAR implementation using this technology should be undertaken.

8 References

[1] David Kirk, Wen-mei Hwu (2010), *Programming Massively Parallel Processors - A Hands-on Approach*, Morgan Kaufmann.

[2] Jason Saunders, Edward Kandrot (2010), *CUDA by Example - An Introduction to General Purpose GPU Programming*, Addison Wesley Professional.

[3] David Blythe (2008), *Rise of the Graphics Processor*, Proceedings of the IEEE Volume 96, Number 5.

[4] B. Neelima, P.S. Raghavendra (2010), *Recent Trends in Software and Hardware for GPGPU Computing: A Comprehensive Survey*, 5th International Conference on Industrial and Information Systems.

[5] Enhua Wua, Youquan Liu (2008), *Emerging Technology about GPGPU*, IEEE Asia Pacific Conference on Circuits and Systems, 2008, pages 618-622

[6] Rob Farber (2008), *CUDA, Supercomputing for the Masses*, Part 1.Dr. Dobb's Journal, April 2008. Available online at http://www.drdobbs.com/architecture-and-design/207200659

[7] Herb Sutter, James Larus (2005), *Software and the Concurrency Revolution*, ACM Queue, Volume 3, Number 7.

[8] Herb Sutter (2005), *The Free Lunch is over: a fundamental turn toward concurrency in software*, Dr. Dobb's Journal 30 (3), Available online at http://www.gotw.ca/publications/concurrency-ddj.htm.

[9] Wen-mei Hwu, Kurt Keutzer (2008), The *Concurrency Challenge*, IEEE Design and Test of Computers, July/August 2008, pages 312-320

[10] Maurice Herlihy (2007). *The Multicore Revolution*, FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science Lecture Notes in Computer Science, 2007, Volume 4855/2007, 1-8

[11] Michael Macedonia (2003), *The GPU Enters Computing's Mainstream*, IEEE Computer 2003, volume 36, issue 10, pages 106-108

[12] George Marsaglia, Arif Zaman, Wai Wan Tsang (1990), *Toward a Universal Random Number Generator*, Statistics & Probability Letters Volume 9, Issue 1, Pages 35-39

[13] F. James (1990), *A review of pseudorandom number generators*, Computer Physics Communications, Volume 60, Issue 3, Pages 329-344

[14] Vadim Demchik (2010), *Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units*, Computer Physics Communications, Volume 182, Issue 3

[15] Home page for GPGPU.org: *General Purpose Computation on Graphics Hardware* http://gpgpu.org

[16] Kayvon Fatahalian and Mike Houston (2008): *A Closer look at GPUs*, Communications of the ACM, 51(10), October 2008.

[17] Nvidia Research Summit 2010 - Poster Listing: Available online at http://www.nvidia.com/object/research_summit_posters_2010.html

[18] G. Moore. (1965): *Cramming more components onto integrated circuits*, Electronics Magazine, Volume 38, Number 8, Available online at: ftp://download.intel.com/research/silicon/moorespaper.pdf

[19] T. C. Weekes, et al. (1989): *Observation of TeV gamma rays from the Crab nebula using the atmospheric Cerenkov imaging technique*, Astrophysical Journal 342, Pages 379–395

[20] Wikipedia definition of Cherenkov radiation (for the interested reader), available online at: http://en.wikipedia.org/wiki/Cherenkov_radiation

[21] D. Heck, J. Knapp, J.N. Capdevielle, G. Schatz, T. Thouw (1998): *CORSIKA: A Monte Carlo Code to Simulate Extensive Air Showers*, Forschungszentrum Karlsruhe Report FZKA 6019

[22] Home page for Corsika - *an Air Shower Simulation Program*: http://www-ik.fzk.de/corsika/

[23] Vadim Demchik and A. Strelchenko (2009), *Monte Carlo simulations on Graphics Processing Units*, arXiv:0903.3053 [hep-lat].

[24] Home page for the CERN Program Library: http://cernlib.web.cern.ch/cernlib/

[25] NVIDIA CUDA C programming Guide Version 3.2, Available online at: http://developer.nvidia.com/object/cuda_3_2_downloads.html

[26] Paul D. Coddington (1997), Random *Number Generators for Parallel Computers*, National HPCC Software Exchange Review, 1.1.

[27] Pierre L'Ecuyer and Richard Simard (2007), *TestU01: A C Library for Empirical Testing of Random Number Generators*, ACM Transactions on Mathematical Software, Volume 33, Issue 4.

9 APPENDIX A – Specification of Test Systems

9.1 Test System1

9.1.1 CPU Specification (cat /proc/cpuinfo)

```
Model name  : Intel(R) Pentium(R) 4 CPU 3.20GHz
Stepping    : 3
Cache size  : 2048 KB
```

9.1.2 Operating System (cat /etc/*release)

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=10.04
DISTRIB_CODENAME=lucid
DISTRIB_DESCRIPTION="Ubuntu 10.04.1 LTS"
```

9.1.3 GPU Driver (cat /proc/driver/nvidia/version)

```
NVRM version: NVIDIA UNIX x86 Kernel Module 260.19.21 Thu Nov 4
20:24:24 PDT 2010
GCC version:  gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5)
```

9.1.4 GPU Specification

```
Device 0: "GeForce GTX 460"
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.20
  CUDA Capability Major/Minor version number:    2.1
  Total amount of global memory:                 1072889856 bytes
  Multiprocessors x Cores/MP = Cores:            7 (MP) x 48 (Cores/MP)
= 336 (Cores)
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.40 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host
threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
```

9.2 Test System2

9.2.1 CPU Specification (cat /proc/cpuinfo)

```
Model name  : Intel(R) Xeon(R) CPU X5670 @ 2.93GHz
Stepping    : 2
Cache size  : 12288 KB
```

9.2.2 Operating System (cat /etc/*release)

```
USE Linux Enterprise Desktop 11 (x86_64)
VERSION = 11
PATCHLEVEL = 1
SGI Foundation Software 2SP1, Build 701r3.sles11-1005252113
SUSE Linux Enterprise Desktop 11 (x86_64)
VERSION = 11
PATCHLEVEL = 1
```

9.2.3 GPU Driver (cat /proc/driver/nvidia/version)

```
NVRM version: NVIDIA UNIX x86_64 Kernel Module 256.35 Wed Jun 16
18:42:44 PDT 2010
GCC version:  gcc version 4.3.4 [gcc-4_3-branch revision 152973] (SUSE
Linux)
```

9.2.4 GPU Specification

```
Device 0: "Tesla C2050"
  CUDA Driver Version:                           3.10
  CUDA Runtime Version:                          3.10
  CUDA Capability Major revision number:         2
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 2817982464 bytes
  Number of multiprocessors:                     14
  Number of cores:                               448
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Maximum sizes of each dimension of a block:    1024 x 1024 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Clock rate:                                    1.15 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
  Compute mode:                                  Default (multiple host
threads can use this device simultaneously)
  Concurrent kernel execution:                   Yes
  Device has ECC support enabled:                Yes
```