A Fast High Quality Pseudo Random Number Generator for nVidia CUDA

W. B. Langdon

Department of Computer Science, CREST centre, King's College, London, WC2R 2LS, UK

Wi11iam.Langdon@kcl.ac.uk

ABSTRACT

Previously either due to hardware GPU limits or older versions of software, careful implementation of PRNGs was required to make good use of the limited numerical precision available on graphics cards. Newer nVidia G80 and Tesla hardware support double precision. This is available to high level programmers via CUDA. This allows a much simpler C++ implementation of Park-Miller random numbers, which provides a four fold speed up compared to an earlier GPU implementation. Code is available via ftp.

Categories and Subject Descriptors: D.2.3 [Coding Tools and Techniques]: Top-down programming

General Terms: Performance

1. INTRODUCTION

Even just a couple of years ago, it was common to disregard the production of pseudo random numbers on graphics hardware [10, 11]. Fortunately last year's CIGPU [5, 8] and previous work [2, sec 7] has helped to dispel this. In particular last year we presented a C++ RapidMind 2 implementation of Park and Miller's minimal standard PRNG [9] which achieved an average of 833 million random numbers per second on an nVidia GeForce 8800 GTX [5, p461]. This has been reimplemented in nVidia CUDA C++ and on an *early engineering* Tesla T10P achieves 3544 million random numbers per second on average. A 4.25 fold increase on the single precision implementation.

Much of the sorry history of software implementations of pseudo random numbers and why we chose Park and Miller's minimal PRNG is described in [5]. A simple serial CPU implementation is given in Figure 1. The large constants used by Park and Miller mean their linear congruent algorithm requires a minimum of 46 bits of precision.

Figure 1 shows the algorithm is quite simple and consequently fast. (A modern Linux PC can generate 27 million random numbers per second [5].)

2. CUDA DOUBLE FLOAT PARK-MILLER

Figure 2 shows a fragment of the C++ CUDA kernel which is run on the graphics card by many execution threads in parallel. Notice that the modulus operation (`%`) on `seed * a` is achieved by replacing `%` by subtracting the smallest integer multiple of `m` which does not exceed `seed * a`. This in turn is found by multiplying `m` by the integer part of `seed * a/m`. However, especially in Tesla double precision, it is faster to multiply by $\mathtt{m}^{-1}$ rather than to divide by `m`.

GECCO'09, July 8–12, 2009, Montreal.
ACM 978-1-60558-505-5/09/07.

```
int intrnd (int& seed) // 1<=seed<m
{
int const a    = 16807;      //ie 7**5
int const m    = 2147483647; //ie 2**31-1
seed = (long(seed * a))%m;
return seed;
}
```

Figure 1: park-miller.cc long int implementation. Multiplication and modulus are used to return a randomised version of the input. By careful choice of `a` and `m` Park and Miller [9] produce an apparently random sequence of integers which uniformly samples the first 2^{31}–2 integers without repeating any.

```
double const a    = 16807;      //ie 7**5
double const m    = 2147483647; //ie 2**31-1
double const reciprocal_m = 1.0/m;

double temp = seed * a;
seed = (int)(temp - m * floor(temp * reciprocal_m));
```

Figure 2: `park-miller_kernel.cu` double precision implementation of Park-Miller CUDA kernel. Notice the modulus operation (cf. Figure 1) has been replaced by subtraction, truncation to integer and double precision multiplication.

3. TESTING

The CUDA kernel is invoked by C++ code running on the host CPU. The Tesla returns a vector of random numbers, each one seeded with a different initial value. These were automatically compared with values given by our original C++ implementation. Both were validated using the method suggested by Park and Miller [9]. Also they were each run up to 2 147 483 647 times and their results confirmed against those at `http://www.firstpr.com.au/dsp/rand31/rand31-park-miller-carta.cc.txt`

4. CONFIGURATION & PERFORMANCE

Using `ebuild merge` we installed 1) nVidia drivers (180.27), 2) CUDA toolkit 2.1 and 3) CUDA SDK 2.10 (1215.2015) on a dual 2.66GHz CPU PC running 2.6.24-gentoo-r3 GNU/Linux with gnu gcc 4.1.2 C/C++ compiler. Default setting, including nvcc compiler optimisation, were used through out.

We used an early release of an nVidia Tesla T10P (192 computation cores $\times$1.08 GHz) which can only give an indication of performance on production Tesla or future releases. CUDA SDK's bandwidthTest program suggests the PC/Tesla combination is capable of 1882 Megabytes per second data transfer rate to the device and 1433 MB/sec back to the host PC.

For ease of comparison with previous work [5] we again generate in parallel up to 4 million streams of PRNG. To estimate the actual costs of data transfers and starting and stoping execution threads we calculate: the next, the next 10 and next 100 random numbers from each random number sequence.

We arranged the CUDA execution threads in one dimensional blocks, which were in turn arranged in a one dimensional grid. The maximum number of active threads is given by the product of the number of threads per block and the number of blocks in the grid. However the number of threads actually running is limited by the number of processing cores, their arrangement, their loading, the availability of data within the Tesla, etc.

As expected, the rate at which random numbers are generated is reduced when the number of threads per block is eight or less. Each block is run on a multiprocessor consisting of eight cores. Therefore at least eight threads per block are need to keep the whole multiprocessor busy. In fact, to allow multiple threads to execute when others are blocked, nVidia recommend at least 16 threads per block. Indeed Figure 3 confirms this and shows performance is reduced when the number of threads per block is ≤ 8. Figure 3 show performance for block size 1, 2, 4, etc., up to 512. (512 is the maximum block size for the Tesla.)

Each block of threads runs on a single multiprocessor. Therefore at least 24 blocks of threads are needed to get the best from the Tesla. Again it appears to be necessary to have more than 24 blocks of threads active to keep the whole Tesla busy. Figure 3 suggests increasing the grid size above 64 does not give much additional performance. Figure 3 shows performance for grid sizes 1, 2, 4, etc., up to 8192.

Figure 3 suggests with only when more than 2048 threads are available will the $8 \times 24 = 192$ cores be fully loaded.

Inspection of the nvcc compiler output (cf. Figure 4) reveals there are ten double precision operations (`f64`) and five 32 bit instructions for each random number. (The 32 bit instructions are not shown in Figure 4.) However run time may be dominated by the three double precision multiplications (`mul.f64`). If we concentrate on the ten double precision operations and multiply by the peak rate at which Park-Miller random numbers are calculated (3544 million/sec) this suggests the Tesla's maximum average double precision performance with our kernel is 35 Gflops.

If the Park-Miller CUDA kernel was really able to fully load the T10P it would be using $192 \times 1.08\,10^9$ clock ticks per second. I.e., $\frac{192 \times 1.08\,10^9}{3544\ 10^6} = 59$ clock ticks per PRNG. If each of the non-`mul.f64` operations takes a tick, this would suggest each `mul.f64` takes in the region of 16 ticks.

5. ADVANTAGES OF CUDA

It is unclear if this is due to CUDA or the different hardware set up but the CUDA/Tesla combination has been much more stable than several earlier Linux PC/GeForce 8800 systems. The system has never hung and the operating system has never been forced to reboot.

```
cvt.rn.f64.u32  %fd1, %r19;
mov.f64         %fd2, 0d40d069c000000000; // 16807
mul.f64         %fd3, %fd1, %fd2;
mov.f64     %fd4, 0d3e00000000200000; //4.65661e-10
mul.f64         %fd5, %fd3, %fd4;
cvt.rmi.f64.f64 %fd6, %fd5;
mov.f64     %fd7, 0d41dfffffffc00000; //2.14748e+09
mul.f64         %fd8, %fd6, %fd7;
sub.f64         %fd9, %fd3, %fd8;
cvt.rzi.s32.f64 %r24, %fd9;
```

Figure 4: Fragment of ptx assembler generated by nvcc for kernel shown in Figure 2. `reciprocal_m` is `4.65661e-10`. There are ten double precision (or mixed format) operations per random number.

The T10P provides 1 Gbytes of storage. Using CUDA all of this can be used in a single operation. I.e., there is no need to split work up into 4 million sized units. E.g., should you wish, it is possible to generate in parallel 250 million independent streams of random numbers and to transfer 250 million random numbers in one go. (Each Park-Miller random number occupies four bytes.)

As well as supporting double precision, CUDA eases lower level access to the GPU or Tesla. E.g. CUDA allows access to the meta-assembly language (ptx) generated by the compiler. In some cases lower level knowledge of the GPU architecture is required. For example RapidMind 2 did not require the programmer to think in terms of cores or multiprocessors. Instead RapidMind divides the work up between the parallel units for the programmer.

6. DISCUSSION

CUDA SDK provides several "quasi random" number generators. These claim about the same performance as Park-Miller. (One is about twice as fast and one about half the speed of our CUDA Park-Miller.) However it is notable that the CUDA implementation does not provide identical results to their own C++ implementation. How "good" a pseudo random number is depends on it usage [4]. We have implemented a PRNG for CUDA and validated it against Park and Miller's minimal standard PRNG [9].

Howes and Thomas [3] implement rather different PRNGs in CUDA and show similar performance. Also they claim speed ups using a "GeForce 8 GPU" (relative to a quad 2.2 GHz PC) of 26 and 59 fold for two financial applications. Gulati and Khatri [1] use a GeForce 8800 GTX to perform Monte Carlo based simulations but of the speed of digital circuits rather than markets. CUDA based PRNG have also been used when simulating biochemical systems [6].

7. CONCLUSIONS

A fast GPU implementation of a pseudo random number generator, meeting Park and Miller's minimum recommendations [9] has been described and benchmarked. It has been implemented in CUDA and demonstrated on an nVidia Tesla. The code is available via anonymous ftp from `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/random-numbers/cuda_park-miller.tar.gz`

Measurements suggest the *early engineering sample* Tesla is at least 130 times faster than running Park-Miller on the host CPU and transferring pseudo random numbers to the GPU (even ignoring the cost of copying the data into the

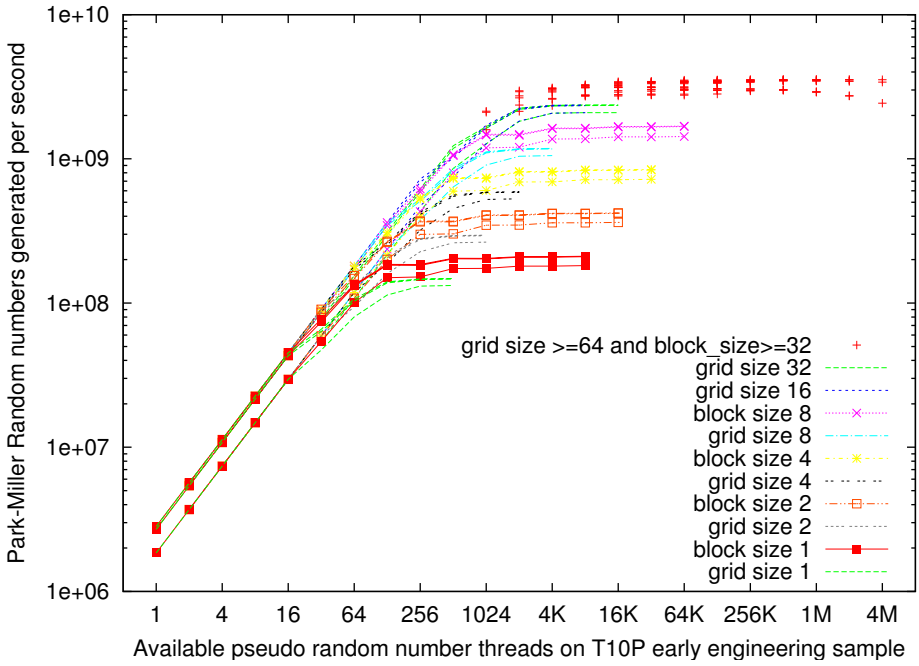

Figure 3: Park-Miller random numbers per second (excluding host-GPU transfer time) on nVidia pre-release Tesla. In the test environment the rate depends upon how effectively the 192 parallel cores can be used. The lower line of each pair refers to calculating just the next random number in the Park-Miller sequence. Therefore these lines include the cost of starting each thread and other overheads. In contrast the upper lines refers to generating either 10 or 100 random numbers together and so better reflects the actual cost of generating the Park-Miller random numbers on the Tesla.

Tesla). Currently single board Tesla are available with up to 240 cores each running at 1.5 GHz, suggesting a further increase in performance of up to 1.74 fold might be possible.

Acknowledgment

The Tesla [7] T10P early engineering sample was supplied by nVidia.

I am greatful for the assistance of Timothy Lanfear, Chris Butler and Sumit Gupta of nVidia, Graham Ashton and William Shaw of KCL, Simon Harding of Memorial and Michal Januszewski of gentoo.org.

8. REFERENCES

[1] Gulati, K., and Khatri, S. P. Accelerating statistical static timing analysis using graphics processing units. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation* (Yokohama), IEEE Press, pp. 260–265.

[2] Harding, S. L., and Banzhaf, W. Fast genetic programming and artificial developmental systems on GPUs. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)* (Canada, 2007), IEEE, p. 2.

[3] Howes, L., and Thomas, D. Efficient random number generation and application using CUDA. In *GPU Gems 3*, H. Nguyen, Ed. nVidia, 2007, ch. 37.

[4] Knuth, D. E. *The Art of Computer Programming*, 2nd ed., vol. 2 Seminumerical Algorithms. Addison-Wesley, 1981.

[5] Langdon, W. B. A fast high quality pseudo random number generator for graphics processing units. In *2008 IEEE World Congress on Computational Intelligence* (Hong Kong) J. Wang, Ed., pp. 459–465.

[6] Li, H., and Petzold, L. R. Stochastic simulation of biochemical systems on the graphics processing unit. Submitted 2007.

[7] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro, 28*, 2 (March-April 2008), 39–55.

[8] Pang, Wai-Man, Wong, Tien-Tsin, and Heng, Pheng-Ann Generating massive high-quality random numbers using GPU. In *2008 IEEE World Congress on Computational Intelligence* (Hong Kong, 1-6 June 2008), J. Wang, Ed., IEEE Computational Intelligence Society, IEEE Press.

[9] Park, S. K., and Miller, K. W. Random number generators: Good ones are hard to find. *Communications of the ACM 32*, 10 (Oct 1988), 1192–1201.

[10] Warden, P. Random numbers in fragment programs, 10 May 2005. Accessed 24 March 2009.

[11] Wong, Tien-Tsin, Wong, Man-Leung, and Fok, Ka-Ling Why current GPU is no good for high-quality random numbers generation? `http://www.cs.cuhk.edu.hk/~ttwong/software/ecgpu/ecgpu.html` epgpu version 0.99. Accessed 24 March 2009.