Random Numbers on GPUs

W. B. Langdon

Mathematical and Biological Sciences
and Computing and Electronic Systems

University of Essex

CIGPU 2008
Introduction

• Artificial Intelligence needs Randomisation
• Implementing randomisation is hard.
• GPU no native support for bit level operations, long integers etc.
• Widespread fear of GPU implementation of random numbers.
• Demonstrate GPU can generate billions of random numbers.
• 400+ speedup v single precision Park-Miller
Need for Random Numbers

• Many Computational Intelligence techniques need cheap randomisation
  – Evolutionary computation: selection and mutation
  – Simulated Annealing
  – Artificial Neural Networks: random initial connection weights
  – Particle Swarm Optimisation
  – Monte Carlo methods, e.g. finance, option pricing
Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

John von Neumann

- History of pseudo random numbers (PRNG) is littered with poor implementations. IBM’s randu described by Knuth as “really horrible”.
- Still true: bug in SUN’s random.c
- Care needed when choosing random method

Knuth
Park and Miller

- Park-Miller big study of linear congruent PRNGs. Fast.
- Suggest “minimal standard” PRNG.
- Uniform integer in 1 to $2^{31}$
- Mandatory PRNG for proposed internet error correction standard.
- Requires 46 bits (Mersenne Twister $\approx 20k$).
- 46 bits typically implemented using long int or double precision. Not available on GPU.
Park-Miller

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

- Next “random” number produced by multiplying current by $7^5$ then reducing to range 1 to $2^{31} - 2$ using modulus $\% m$
- Multiplication produces 46 bit result
- All calculations use integers
GPU Park-Miller

- C++ implementation under RapidMind
- GPU float only single precision
- Use Value4f (vector of 4 floats) to store and pass random numbers.
- 31 bits of Park-Miller broken into 4 bytes. Each byte stored as float. So no rounding problems.
- Value4f native GPU data type.
exactmul $7^5 \times \text{Value4f} \rightarrow \text{Value5f}$

```cpp
exactmul(float f, float in[4], float out[5]) {
    out[0]=0;
    for(int i=0;i<4;i++) {
        const float t=in[i]*f;
        out[i] += t; //Max value 16807+16807×255
        out[i+1] = floor(out[i]/256);
        out[i] = int(out[i])%256;
    }
}
```

By performing multiplication a byte at a time calculation can be done with float
Parallel RapidMind exactmul

$7^5 \times \text{Value4f} \rightarrow \text{Value5f}$

```c
inline void exactmul(const Value1f f, const Value4f in, Value<5,float>& out) {
    //RM_DEBUG_ASSERT(f <= Value1f(16807));
    out[0] = 0;
    for(int i=0; i<4; i++) {
        out[i] += round(in[i]*f);
        out[i+1] = floor(out[i]/Value1f(256));
        out[i] = round(Value1i(out[i])%256);
    }
}
```

multiplication a byte at a time so can be done with float. Round to ensure exact integer values.
Value used to represent 46 bits

- 8 bits
  - Least significant bit
  - 0–7
  - 8–15
  - 16–23
  - 24–31
  - 14 bits
    - Most significant bit
    - 32–45
prng = (prng\times 7^5)\% 2147483647

• After exactmul need to reduce modulus 2147483647 but 2^{31}-1 can not be represented exactly by float.

• Replace % by finding largest exact multiple of 2^{31}-1 which does not exceed prng\times 7^5 then subtract it from prng\times 7^5.
  – Avoids long division

• This gives the next Park-Miller pseudo random number.
Finding largest multiple of $2^{31}-1$ not exceeding $\text{prng} \times 7^5$

- Find (approx) $(\text{prng} \times 7^5)/(2^{31}-1)$
- Refine approximation
- Multiply exact divisor by $2^{31}-1$
- Obtain next PRNG by subtracting exact multiple of $2^{31}-1$ from $\text{prng} \times 7^5$.
- Multiply and subtraction can be done exactly (using trick of splitting long integer into 8-bit bytes and storing these in floats).
Finding Largest Divisor 1

Value1i approxdiv = floor(prng*a/m);
Value1i comp = -1; // loop at least once
FOR(nul,comp<0,nul) {
    exactmul(Value1f(approxdiv),M,multiM);
    comp=comp5(out,multiM); //nb out=a*Prng
    approxdiv--;
}ENDFOR

a= 7^5 M = 2^{31}-1

• For loop used to decrement approxdiv until multiM=approxdiv*(2^{31}-1) ≤ prng*7^5
• Mostly only loop only used 1 or 2 times.
Finding Largest Divisor 2

```c
exactsub5(out,multiM,Prng); // prng = out-multiM
FOR(nul,comp4(Prng,M)>=0,nul) {
    exactsub4(Prng,M,Prng); // prng=prng-m;
}ENDFOR

a= 7^5 M = 2^{31}-1
```

- In case approxdiv was too low the FOR loop is used to reduce the new PRNG by repeatedly subtracting $2^{31}-1$ until it is below $2^{31}-1$. 

Comp4 using RapidMind

```cpp
inline Value1i comp4(const Value4f a, const Value4f b) {
    return cond(a[3]>b[3],Value1i(+1),
            cond(a[3]<b[3],Value1i(-1),
                cond(a[2]>b[2],Value1i(+1),
                    cond(a[2]<b[2],Value1i(-1),
                        cond(a[1]>b[1],Value1i(+1),
                            cond(a[1]<b[1],Value1i(-1),
                                cond(a[0]>b[0],Value1i(+1),
                                    cond(a[0]<b[0],Value1i(-1),Value1i(0)))))))));
}
```

Use GPU cond to compare most significant parts of a and b first
Exactsub5 using RapidMind

- Operate on local copies of inputs to avoid side effects on calling code.
- Requires \( a \geq b \) and \( a-b \) fits in 4 bytes
- Subtract \( B[i] \) from \( A[i] \). Use round to force integer
- If \( A[i] < B[i] \) “borrow” 256 from \( B[i+1] \).
- No negative values

```cpp
//nb a>=b
for(int i=0;i<4;i++) {
    B[i+1] = cond(A[i]<B[i],round(B[i+1]+Value1f(1)),B[i+1]);
    out[i] = round(A[i]-B[i]);
}
```
Validation

- RapidMind GPU and two PC version of Park-Miller were each validated by generating at least the first 100 million numbers in the Park-Miller sequence and comparing with results in Park and Miller’s paper and www.
In test environment, with ≥ 8192 threads the 128 stream processors give peak performance. I.e. ≥16 active threads per SP. Or ≥512 threads per G80 8SP block.
Performance

• nVidia GeForce 8800 GTX (128 SP)
• 833 $10^6$ random numbers/second
• 44 times double precision CPU (2.40Ghz)
• More than 400 times single precision CPU
• Estimate 90 GFlops (17% max 518.4 nVidia claim)
Discussion

- 90GFlops too high?
- Test harness semi-realistic.
- GPU application, PRNG just a small part, but avoids communication with CPU.
- Main bottle neck is access to GPU’s main memory.
- PRNG faster if use on-chip memory but application may want this for other reasons.
- Importance of many threads (min 512).
Conclusions

• Cheap randomisation widely needed but often poorly implemented.
• Fear of PRNG on GPU (said GPU cant do)
• Park-Miller fast but needs more than float
• GPU implementation meets Park and Miller’s minimum recommendation.
• RapidMind C++ Code available via ftp.
• Up to 833 million pseudo random numbers per second.
END
Questions

• Code via ftp

• gpgpu.org GPGppgpu.com rapidmind.com