Generating Massive Amount of High-Quality Random Numbers using GPU

Wai-Man Pang, Tien-Tsin Wong, Pheng-Ann Heng

The Computer Science and Engineering Department
The Chinese University of Hong Kong

IEEE WCCI CIGPU 2008
Pseudo-random number generator (PRNG)

- Provide uniform random numbers
- Example: `rand()` in C
- Important for stochastic algorithms
  - Evolutionary Computing
  - Photon-mapping rendering
- Huge Amount
- Speed
- Quality
  - Poor randomness $\rightarrow$ slow convergence
PRNG for Stochastic Rendering

- Artifact for poor quality PRNG
PRNG for Stochastic Rendering

- From High quality PRNG
Some common PRNG

- linear congruential generator (LCG)
  - \( R_{n+1} = aR_n + b \pmod{m} \)
- lagged Fibonacci generator
  - \( R_n = R_{n-j} \# R_{n+k} \pmod{m} \) (where \(#\) is a binary operator)
- High precision integer arithmetic
- Cannot fit in all GPU
PRNG on GPU

- Cellular Automata-based PRNG [Wolfram]
- No high precision integer arithmetics
- Homogeneous cell operation and connectivity
- Quality
  - Configure to produce high quality random sequence
CA-based PRNG

- Array of connected cells with homogeneous behavior
- Each Cell have a state and a common cell equation
- Cell Equation:

  \[ c_i^g = \phi(c_{i+n_0}^{g-1}, c_{i+n_1}^{g-1}, ..., c_{i+n_j}^{g-1}) \]

\[ \Phi(X) \]

Previous state values from neighbors

Output state value
Mechanism

- 4 Cell, Connectivity (-1,2)
- Cell Equation: step(1, 3 - c1 - 2*c2)

Cell D: 1  
Cell C: 0  
Step(1, 3 - 1 - 2*0)
Mechanism (cont’)

A B C D
1 0 1 1

random number generated 111

A B C D
0 0 1 1

random number generated 011
GPU Implementation Issue

- Cell resembles texel in GPU
- 64 cells and 4 connected CA PRNG for 32-bits random number
- Cell equation evaluation
  - Fast table lookup
  - 4 connectivities = 4 input, $2^4 = 16$ possible output
- Reorganize bits
  - Bits in a random number is scattered among texels
  - Output floating point value $f$

\[
f = \left( \left( \left( \left( \frac{r_0}{2} \right) + r_1 \right) / 2 + \ldots + r_{31} \right) / 2 \right)
\]

- $r_i$ is the $i$-th bit in the random number
float4 caprng( in half2 coords: TEX0, in const uniform samplerRECT cells): COLOR0
{
    float2 Connector; float4 newState; float4 neighborStates[4]; int i;
    for (i = 0; i < 4; i++)
    {
        Connector.x = fmod(coords.x - connectivity(i), CA SIZE);
        Connector.y = coords.y;
        neighborStates[i] = round(texRECT(cells, Connector));
    }
    // cell equation evaluation
    newState.x = celleqn(neighborStates);
    return newState;
}

float4 pack(in half2 index : TEX0, in const uniform samplerRECT cells): COLOR0
{
    int i; float4 outbits; float4 states; float2 texindex; outbits = 0;
    // packing all 32 bits
    for (i = 0; i < 32; i++)
    {
        texindex.x = i*2+1;
        texindex.y = index.y;
        states = texRECT(cells, texindex);
        outbits += states;
        outbits /= 2;
    }
    return outbits;
}
Parallelized PRNG

- Fully utilize $4096 \times 4096$ texels (7800 GTX)
- Each cell occupies single bit in texel
- Why not pack more inside each texel?
  - Fully utilize the mantissa part of the texel
- $23 \times 4$ random sequences simultaneously.
- Combine 2 schemes: $64 \times 4096 \times 92$ PRNGs

Cells Texture

PRNG1: 1 1 0 0
PRNG2: 0 1 1 0
PRNG3: 0 1 0 1
Optimize for Quality

- **Genetic Algorithm**
  - CA base PRNG configuration with best quality
- **Initialize candidates**
  - Encoded cell equation and connectivities
  - $2^n + n$ bits
- **Evaluate candidates by objective function**
- **Generate next generation**
  - Crossover
  - Mutation
- **Repeat until excess certain threshold**
Objective Function

- Objective function
  \[ \text{objective} = w_0 \times e + w_1 \times \varphi \]
- \(w_i\) is the weighting
- \(e\) is the n-bit entropy

\[
e = \frac{-\sum_{i=0}^{2^n-1} p_i \log p_i}{n}
\]

- \(\varphi\) is the result of Diehard test
Objective Function (cont’)

- **Diehard test**
  - 14 tests (e.g. birthday spacing, GDC test, etc.)
  - Chi-square

\[ \chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i} \]

- **Overall p-value**
  - Chi-square test on all p-values with Gaussian distribution

- **Best 4 connected, 64 Cells CA PRNG**
  - Connectivity (56,2,21,49)
  - Cell equation in tightly packed format
    (1001100110100101)
Convergence

Control
10,000 photons

e=0.2673 \ \varphi=0.0

Generation 1

e=0.5852 \ \varphi=0.0

Generation 2

e=0.5944 \ \varphi=0.0

Generation 4

e=0.5944 \ \varphi=0.0

Generation 8

e=0.9464 \ \varphi=0.143

Generation 11

e=0.9514 \ \varphi=0.3513
## Performance

- **Performance compare with CPU**
- **Single PRNG**
- **1,000 Parallel PRNG**

<table>
<thead>
<tr>
<th>Random numbers generated</th>
<th>GPU CA-PRNG</th>
<th>Software CA-PRNG</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>0.004s</td>
<td>0.003s</td>
</tr>
<tr>
<td>100,000</td>
<td>0.042s</td>
<td>0.042s</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.942s</td>
<td>0.425s</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10.081s</td>
<td>4.374s</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100.082s</td>
<td>43.083s</td>
</tr>
<tr>
<td>100,000,000</td>
<td>31.875s</td>
<td>430s</td>
</tr>
</tbody>
</table>
**Conclusion**

- CA architecture PRNG is highly suitable for GPU
- Parallel PRNG on GPU
- Optimization for quality
- A high quality and high performance gain
- Future works
  - Support of variable precision random sequence
  - Experiment with Evolution Computing applications
End

Thanks for your attention
Reference: