Simulational Methods

Fast Random Number Generation Using 128-Bit Multimedia Extension Registers on Pentium Class Machines

BORKO D. STOŠIĆ

Departamento de Estatística e Informática, Universidade Federal Rural de Pernambuco, Dois Irmãos, Recife-PE, Brasil

In this article, it is shown how 128-bit SSE2 multimedia extension registers, present in Pentium IV class 32-bit processors, may be used to generate random numbers at several times greater speed than when regular general purpose registers are used. In particular, a 128-bit algorithm is presented for the Marsaglia MWC1616 generator from the DIEHARD battery of random number generator tests, and its performance is compared to that of the conventional approach.

Keywords Multimedia; Pentium processors; Random numbers.

Mathematics Subject Classification ▪

1. Introduction

Scientific computing has seen some impressive development over the last couple of decades, made possible by the exponential advent of cheap computer resources: what was double a decade ago only on a supercomputer, today can be implemented on a laptop. On the other hand, the appetites of scientists using computational techniques have grown in parallel, so that algorithms used nowadays often require weeks (or months) of continuous processing, same as (orders of magnitude more modest) algorithms used a decade or two ago. While we can certainly expect the performance of processors to continue improving (driven by general market requirements), it is a fact that during a single clock tick of a (currently common) 3 GHz processor, the light traverses only 10 cm—roughly twice the linear dimension of the actual-processor chip (this is actually an overestimate, as electrical signals...
propagate through matter at speeds lower than the speed of light in vacuum of
$3 \times 10^8$ m/s). Significant further increase in clock times is therefore closely related
to reducing the physical dimensions of the chip, and currently, parallel computing
seems to be the most promising way out of these physical limitations.

In fact, the Intel processor manufacturer has already adopted this strategy
some years ago, and most personal computers today already contain parallel
processing hardware capability (Intel, 2006). More precisely, starting with Pentium
II processors, SIMD (Single Instruction Multiple Data) parallel computation on
eight new internal 64-bit registers (called MM0-MM7) was introduced (this standard
was named MMX, an abbreviation for Multimedia Extension). The concept was
further improved in Pentium III processors by introducing another 8, 128-bit
registers (called XMM0-XMM7) with a corresponding instruction set (the standard
was named SSE – Streaming SIMD Extensions), and in Pentium IV the SSE2
standard was implemented with an improved instruction set. Finally, in the Pentium
IV processor 3.40 GHz, supporting Hyper-Threading Technology, SSE3 standard
was introduced with additional 13 instructions. The current SSE2/3 standard works
with packed data (two double precision, four single precision, or 16/8/4/2 integers
of 8/16/32/64 bits each), where a single instruction is simultaneously executed on
the packed variables. When dealing with single precision floating point numbers or
32-bit integers, this architecture yields (roughly) 4 times the performance of regular
SISD (Single Instruction Single Data) processing on the same processor.

It seems that the above developments have not been widely recognized by
the scientific community, while evidently they may prove crucial in a number of
situations where performance is critical. One such example is the (pseudo) random
number generation, which often proves to be the bottleneck in high-performance
simulations (such as Monte Carlo, Simulated Annealing, Bootstrap, etc.) requiring
high precision, when high-periodicity and low correlation of the random number
sequence is required.

In this work, a 128-bit algorithm is presented for the Marsaglia’s MWC1616
generator (Marsaglia, 1985, 1996, 1999) with a period of roughly $2^{59}$, which passes
all of the tests in the stringent DIEHARD suite of random number generator
tests (Marsaglia, 1996). Performance of this algorithm is then compared with
the conventional approach, both using function calls and C language macros. In
the following section, first a brief overview of the MWC1616 generator is given,
followed by a straightforward implementation in C, then a better macro version,
and finally, the 128-bit parallel algorithm. The subsequent section deals with the
performance testing results of the presented algorithms, obtained on a Celeron
1.6 GHz processor. Finally, the conclusions are drawn.

2. The MWC1616 Uniform Deviates

2.1. The MWC1616 Generator

The MWC1616 uniform (pseudo)random number generator (Marsaglia, 1985, 1996,
1999) concatenates results of two 16-bit MWC (Multiple With Carry) generators to
produce a 32-bit result. The two 16-bit generators have the form

$$x_{n+1} = [ax_n + c_n] \mod 2^{16},$$
$$y_{n+1} = [by_n + d_n] \mod 2^{16},$$

(1)
where \( a \) and \( b \) are multipliers (a good choice is \( a = 18,000 \) and \( b = 30,903 \), a table of suggested values can be found in Marsaglia, 1996), \( c_n \) and \( d_n \) are corresponding 16 bit overflow values (carry) resulting from 16-bit multiplication at level \( n \), and the symbol “mod” indicates modulus operation. The period of this generator is given by \((a \times 2^{15} - 1)(b \times 2^{15} - 1)\) (Marsaglia, 1996), yielding over \( 2^{69} \sim 6 \times 10^{17} \) for the choice \( a = 18,000 \) and \( b = 30,903 \). The generator is easily implemented in C by using unsigned long 32-bit integers to store \( x_n \) and \( y_n \) in the low words and carry values \( c_n \) and \( d_n \) in the high words, using only two statements

\[
\begin{align*}
\text{x} &= \text{a} \times (\text{x} \& \text{0xFFFF}) + (\text{x} >> 16); \\
\text{y} &= \text{b} \times (\text{y} \& \text{0xFFFF}) + (\text{y} >> 16);
\end{align*}
\]

while the 32-bit integer to be returned (which represents the member of the random sequence), is calculated by concatenating the two low words with the statement

\[
(\text{x} << 16) + (\text{y} \& \text{0xFFFF}).
\]

Finally, before running the generator one needs to initialize the two 16-bit seeds and the initial carry values, by choosing a value for \( 0 < x_0 \leq 2147483647 \) and \( 0 < y_0 \leq 2147483647 \) (2147483647 is decimal for 0x7FFFFFFF, the 16-bit seed is stored in the low word, and initial carry value in the high word).

### 2.2. C Implementation

The most straightforward full implementation in C programming language of the MWC1616 generator given by Eq. (1), is given below:

```c
static unsigned long x = 1, y = 2;

void seed(unsigned long x0, unsigned long y0)
{
    x = x0;
    y = y0;
}

unsigned long MWC1616()
{
    x = 18000 * (x & 0xFFFF) + (x >> 16);
    y = 30903 * (y & 0xFFFF) + (y >> 16);
    return (x << 16) + (y & 0xFFFF);
}
```

where the initial seed values have been set to \( x_0 = 1 \), \( y_0 = 2 \), and \( c_0 = d_0 = 0 \). The seed() function should be called on initialization, before the first call to MWC1616(), to specify the origin of the random number sequence.

While extremely simple and efficient (in comparison with many other random number generators), this implementation suffers from the fact that each function call is accompanied by function prologue and epilogue overhead (sequences of assembly instructions inserted by the compiler on the beginning and the end, respectively, of
any function call). It is rather more efficient to replace the MWC1616() function call
by a C language macro, as follows:

```c
#define xnew (x = (18000*(x&0xFFFF) + (x >> 16)))
#define ynew (y = (30903*(y&0xFFFF) + (y >> 16)))
#define MWC1616 ((xnew << 16) + (ynew&0xFFFF))
```

This approach dispenses with the function call overhead (the machine language
opcode is literally inserted by the compiler at all places where the macro is called),
while implementing exactly the same sequence of operations. The function seed()
need not be replaced with a macro, as it is called only on rare occasions (normally
only upon initialization).

2.3. **SSE2 128-Bit Implementation**

In order to implement the MWC1616 generator in 128-bit arithmetic, in what
follows the SSE2 Pentium IV standard shall be used, as this is currently probably
the most widespread situation, and it seems that the SSE3 extensions found in newer
processors do not provide any additional functionality relevant for the current
implementation.

Unfortunately, in order to implement the MWC1616 generator using the 128-
bit XMM registers and the SSE2 instruction set, one needs to resort to assembly
language programming. The problem is that these are highly specialized features
of the processor, introduced mainly for the purpose of multimedia streaming and
high-performance graphics processing. Consequently, the high-language compiler
manufacturers have not found an interest in incorporating these features in any of
the high-level languages (such as C or Fortran), with the exception of the Intel C
compiler. As the current author had no access to this particular compiler at the time
of writing this article, inline assembly code was used within the Microsoft Visual
Studio 6.0 environment, using the cl 32-bit C/C++ compiler. In order for this
environment to recognize the SSE2 extensions together with the XMM register set,
first the Intel SSE2 processor pack had to be installed. It should be mentioned here
that the use of the Intel C compiler essentially requires identical effort in writing
SSE2 code as assembly programming, as the C language instructions have practically
one-to-one correspondence with their assembly language equivalents.

Although most personal computers nowadays fall into Pentium IV class, before
embarking on SSE2 assembly programming one should check for the existence of
these features on the processor to be used. This can be accomplished (in assembly)
by loading the eax register with value 1, issuing a CPUID instruction, and then
examining the edx register: if the bit 26 is set, the processor supports SSE2
extensions. The following function in C with inline assembly may be used to
implement this test (here the “//” symbol indicates a comment from the current
position to the end of the line):

```c
int SSE2Available()
{
int available = 0;
_asm
```
If the processor does not support SSE2 extensions the algorithm described in the rest of this article will not work. It should be mentioned here that the check for support of earlier standards SSE and MMX may be performed by examining bits 25 and 23, respectively, of edx after CPUID instruction call, while support of the latest SSE3 standard is returned in bit 0 of the ecx register.

Let us now turn to the actual 128-bit implementation of the MWC1616 generator. First, as the current implementation deals with fourfold parallel instructions, one needs to initialize eight seeds (rather than only two). Next, manipulation of registers is generally faster than operations performed between registers and memory, so constant arrays are first declared and initialized, to be loaded in scratch xmm registers on startup. The following code snippet was used to declare and initialize variable arrays and constants:

```c
static unsigned int x[4] = 1,1,1,1,
y[4] = 2,2,2,2,
r[4],
mask[4] = 0xFFFF,0xFFFF,0xFFFF,0xFFFF,
mul1[4] = 18000,18000,18000,18000,
mul2[4] = 30903,30903,30903,30903;

_asm
{
    movdqa xmm0, x //load array x into xmm0
    movdqa xmm1, y //load array y into xmm1
    movdqa xmm5,mask //load mask
    movdqa xmm6,mul1 //load first multiplier
    movdqa xmm7,mul2 //load second multiplier
},
```

where the SSE2 instruction “movdqa” loads values of variable arrays \( x \) and \( y \) into registers xmm0, xmm1, and constant arrays mask, mul1 and mul2 into registers xmm5, xmm6, and xmm7, respectively, for posterior use. Note that the arrays \( x \) and \( y \) have been initialized with same seeds for all four indices for testing purposes (to verify whether all the four adjacent 32-bit blocks yield the same result), in a real application all four pairs should be initialized to distinct values. More precisely, each of the four seeds of the first four MWC 16-bit generator group has been set to value 1 (low words of \( x[i] \), \( i = 1, \ldots, 4 \)), each of the four seeds of the second generator group has been set to value 2 (low words of \( y[i] \), \( i = 1, \ldots, 4 \)), and all the initial carry values have been set to 0 (high words of \( x[i] \) and \( y[i] \), \( i = 1, \ldots, 4 \)).
A single update of each of the MWC 16-bit generators can be broken down in the following elementary steps on a 32-bit block $x$:

(i) extract the low word (as $x \& 0xFFFF$);
(ii) multiply the result with the constant multiplier;
(iii) extract carry from $x$ (as $x \gg 16$); and
(iv) add results of (ii) and (iii),

and the MWC1616 generator may be implemented with SSE2 instructions as follows:

```asm
asm {
    movdqa xmm2, xmm0 //make a copy of x
    psrld xmm2, 10h //x >> 16 in xmm2
    //now find a*(x&0xFFFF)
    //in current example, a = 18000 = 0xmm1650:
    andps xmm0, xmm5 //x&0xFFFF
    movdqa xmm3, xmm0 //make a copy
    pmullw xmm0, xmm6 //multiply, save low word
    pmulhuw xmm3, xmm6 //multiply, save high
    pslld xmm3, 10h //high result << 16
    orps xmm3, xmm0 //low OR high
    //a*(x&0xFFFF) now (finally) in xmm3
    paddd xmm2, xmm3 //a*(x&0xFFFF) + (x >> 16)
    movdqa xmm0, xmm6 //make a copy
    //now second generator...
    movdqa xmm4, xmm1 //make a copy of y
    psrld xmm4, 10h //y >> 16
    //find b*(x&0xFFFF), where b = 30903 = 0x78B7:
    andps xmm1, xmm5 //y&0xFFFF
    movdqa xmm3, xmm1 //make a copy
    pmullw xmm1, xmm7 //multiply, save low word
    pmulhuw xmm3, xmm7 //multiply, save high
    pslld xmm3, 10h //high result << 16
    orps xmm3, xmm1 //low OR high
    //b*(x&0xFFFF) in xmm3
    paddd xmm3, xmm4 //b*(y&0xFFFF) + (y >> 16)
    movdqa xmm1, xmm3 //copy new value to y
    andps xmm3, xmm5 //and with 0xFFFFh
    paddd xmm2, xmm3 //(x << 16) + (y&0xFFFF)
    //random number in xmm2
    movdqa r, xmm2 //save result in array r
}
```

The principal problem that was encountered when implementing these steps in parallel, on the four adjacent 32-bit blocks stored in the XMM registers, was to perform multiplication in step ii). Namely, the SSE2 instruction set does not contain an adequate instruction to multiply 16-bit values in the low words of the four
adjacent blocks (the high words are by definition zero, as both the current value and
the multiplier are 16-bit integers), and store the results in the same 32-bit blocks.
Instead, two distinct multiplication instructions were used for multiplying 16-bit
blocks, and then storing first the low, and then the high 16-bit result, as exemplified
by the following code snippet:

```assembly
andps xmm0, xmm5  // x & 0xFFFF
movdqa xmm3, xmm0  // make a copy
pmullw xmm0, xmm6  // multiply, save low word
pmulhuw xmm3, xmm6  // multiply, save high
pslld xmm3, 10h  // high result << 16
orps xmm3, xmm0  // low OR high
```

Here, the value (x & 0xFFFF) in the xmm0 register is first copied to xmm3, the two
copies are then individually multiplied by the constant previously loaded into xmm6,
where first low word and then high word results are stored. Next, the high result is
shifted 16 places to the left, and finally OR-ed with the low word result, to yield the
final product values.

3. Performance Tests

The three different implementations of the MWC1616 generator, all yielding
exactly the same sequences (given the same seeds) of uniform deviates, have been
tested using a Celeron 1.6 GHz processor, on a Toshiba Satellite laptop computer.
The tests were performed for all three implementations, both with and without
compiler optimization (flags /Od and /O2 of the cl compiler). Timing results (per
uniform deviate) performed by averaging over 20,000 blocks of 50,000 deviates, are
summarized in Table 1.

It is seen from Table 1 that compiler optimization has a significant impact
only in the case of a straightforward function call (in the case of SSE2 compiler
optimization affects only the outer loops, while inline assembly sequences are
preserved). The standard deviation over the 20,000 runs (of 50,000 deviates each) is
presented to indicate the extent to which the background activity of the operating
system affects the execution time. While relative fluctuations are largest in the SSE2
case, it should be noted that a single cycle (or clock tick) of a 1.6 GHz machine
has a duration of 0.625 nanoseconds: roughly 10 cycles are used per deviate, with

<table>
<thead>
<tr>
<th>Version</th>
<th>Debug</th>
<th>σ_D</th>
<th>Optimized</th>
<th>σ_O</th>
</tr>
</thead>
<tbody>
<tr>
<td>C function</td>
<td>58.47</td>
<td>2.94</td>
<td>48.56</td>
<td>3.14</td>
</tr>
<tr>
<td>C macro</td>
<td>31.15</td>
<td>2.66</td>
<td>29.19</td>
<td>2.40</td>
</tr>
<tr>
<td>SSE2</td>
<td>5.72</td>
<td>1.01</td>
<td>5.34</td>
<td>0.92</td>
</tr>
</tbody>
</table>
a standard deviation of 2 cycles. The fact that 24 SSE2 assembly instructions
constituting the 128-bit implementation of the MWC1616 generator are on average
executed in only 10 cycles per deviate, is the result of the fact that four uniform
deviates are calculated in parallel. More precisely, the 24 instructions in the above
code are executed in $5.72 \times 4 = 30.88$ nanoseconds, yielding four uniform deviates.
Therefore, an effective average of 1.3 (fourfold SIMD) instructions per cycle is
achieved even as a total of four multiplication instructions (generally requiring more
time then other instructions) and three addition instructions were used, which may
be attributed to the pipelining architecture of the processor (4), representing another
aspect of the parallelism paradigm implemented by the Pentium IV processors:
instructions are processed several at a time, similar to a factory assembly line.

4. Conclusion

In conclusion, it follows from the presented algorithms and results that scientific
implementations of 128-bit SSE2 extensions require some “ugly” programing
practice: not only one has to resort to assembly language programing, but the
available instruction set is also somewhat unusual, and far from being complete. On
the other hand, the impressive 500% gain over the optimized C code (or for that
matter straight SISD assembly) may justify such coding practice for time critical
applications, in fact, in certain situations it may make all the difference between an
unfeasible and a feasible problem (an application that would run five months may
be considered unfeasible, and the one that would run a single month, feasible).

Finally, it should be stressed that the application to random number generation
presented in this work by no means represents the only possible application for
scientific computing, many other time critical algorithms (e.g., linear algebra matrix
manipulation) may benefit from this approach (SSE2/3 extensions are not limited
to integer arithmetic, both single and double precision floating point algebra is
supported). A C library with the current implementation of MWC1616 uniform
deviates (which does not require assembly programing nor installation of the
processor pack, unless one wants to modify and recompile the library from source),
can be obtained from the author upon request.

References

Marsaglia, G. (1985). Keynote address: a current view of random number generators,
proceedings, computer science and statistics. 16th Symp. Interface. Atlanta: Elsevier, GA.
Marsaglia, G. (1996). The Marsaglia random number CDROM, with the DIEHARD
Battery of tests of randomness. Department of Statistics, Florida State University,
http://stat.fsu.edu/geo/diehard.html, last seen in October, 2006.
messageID=1524805&tstart=0, last seen in October, 2006.