Vectorization of Philox CBRNG on AVX2, AVX512 and Extending Agner FOG's Vector Class Library for ARM NEON Support

Yigit Demirag

Advisor: Dr. Sandro Wenzel

Outline

1. Vectorization of Philox CBRNG *
2. ARM NEON Support for Agner FOG's Vector Class Library **

* http://www.thesalmons.org/john/random123/

* * http://www.agner.org/optimize/vectorclass.pdf

CERN

Part I: Vectorization of Philox CBRNG • Project Description

- CBRNGs are widely used at CERN, especially in MC Simulations in GEANT4 and ROOT.

- PRNGs are deterministic algorithms in form of

```
uint_64t someRandomNumber = CBRNG(uint64_t key, uin64_t counter)
```

- Philox is a SP Network. S box is a simple Feistel function with 72 rounds 64-bit [XOR, MUL]

$$L' = B_k(R) = mullo(R, M)$$
$$R' = F_k(R) \oplus L = mulhi(R, M) \oplus k \oplus L$$

CERN

Part I: Vectorization of Philox CBRNG • The Mulhilo Bottleneck

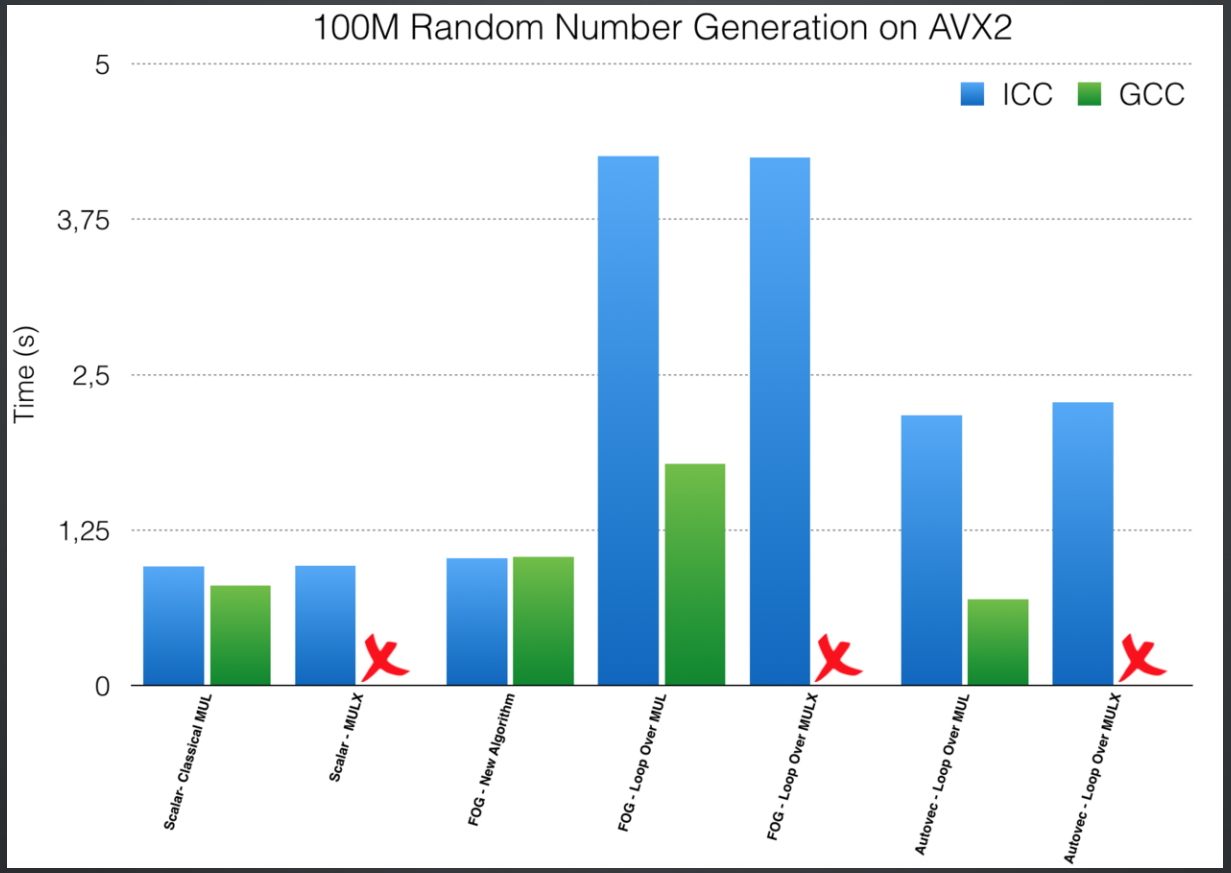
- The main problem in vectorization: 64-bit widening multiplication causing bottleneck.

```
static __inline__ uint64_t mulhilo64(uint64_t a, uint64_t b, uint64_t* hip) {
    __uint128_t product = ((__uint128_t)a)*((__uint128_t)b);
    *hip = product>>64;
    return (uint64_t)product;
}
```

- There is no 64b x 64b -> 128b arithmetic as a vector instruction. Nor is there a vector mulhi type instruction (high word result of multiply).
- Haswell's MULX instruction was a scalar candidate (No effect on register flags, flexible register use).
- Karatsuba Multiplication Algorithm ? (Efficient for much bigger multiplications.)
- Or a new multiplication algorithm designed for SIMD?

CERN

Part I: Vectorization of Philox CBRNG



CERN

Part I: Vectorization of Philox CBRNG • The New Algorithm

```
void muldwu1_AVX2(__m256i x, __m256i y, __m256i *lo, __m256i *hi) {
    __m256i lomask = _mm256_set1_epi64x(0xffffffff);

    __m256i xh     = _mm256_shuffle_epi32(x, 0xB1);     // x0l, x0h, x1l, x1h
    __m256i yh     = _mm256_shuffle_epi32(y, 0xB1);     // y0l, y0h, y1l, y1h

    __m256i w0     = _mm256_mul_epu32(x,  y);           // x0l*y0l, x1l*y1l
    __m256i w1     = _mm256_mul_epu32(x,  yh);          // x0l*y0h, x1l*y1h
    __m256i w2     = _mm256_mul_epu32(xh, y);           // x0h*y0l, x1h*y0l
    __m256i w3     = _mm256_mul_epu32(xh, yh);          // x0h*y0h, x1h*y1h

    __m256i w0l    = _mm256_and_si256(w0, lomask);      //(* Not required at AVX512)
    __m256i w0h    = _mm256_srli_epi64(w0, 32);

    __m256i s1     = _mm256_add_epi64(w1, w0h);
    __m256i s1l    = _mm256_and_si256(s1, lomask);
    __m256i s1h    = _mm256_srli_epi64(s1, 32);

    __m256i s2     = _mm256_add_epi64(w2, s1l);
    __m256i s2l    = _mm256_slli_epi64(s2, 32);         //(* Not required at AVX512))
    __m256i s2h    = _mm256_srli_epi64(s2, 32);

    __m256i hi1    = _mm256_add_epi64(w3, s1h);
            hi1    = _mm256_add_epi64(hi1, s2h);

    __m256i lo1    = _mm256_add_epi64(w0l, s2l);        //(* Not required at AVX512))
    //__m512i lo1    = _mm512i_mullo_epi64(x,y);          //alternative AVX512 inst.

    *hi = hi1;
    *lo = lo1;
}
```

" Alternative instruction existed only on AVX512, we could have advantage of vectorization at Xeon Phi.

CERN

Part I: Vectorization of Philox CBRNG • AVX512(MIC) Implementation

- Philox will benefit from larger vector registers in AVX512. Our goal was to provide an implementation that can be used and benchmarked on **the next generation hardware**, specifically on **Intel's Knights Landing(KNL)** architecture.
- FOG's VCL supports only KNL, but OpenLab contribution for Knights Corner(KNC) exists.
- I vectorized Philox library both for KNC and KNL, via both library and macros. But, since the new vectorized multiplication algorithm requires specific KNL instructions, we haven't benchmarked this work yet.

CERN

Part-II : ARM Support for Agner FOG's VCL • Project Description

- We wanted to provide an easy-to-use ARM NEON support for Agner FOG's Vector Class Library.
- The aim was to compile the same code for both Intel's SSE and ARM architecture.
- We concentrated on single and double precision floating point operations of ARM NEON (v7 and v8), which has 128-bit registers that can be used as 64bx2 or 32bx4 vector operations.

> *An ARM processor uses ~ 5-15 watts, a traditional computer can use up to 80 watts. Server farms have on the order of tens of thousands of computers and so this translates to significant savings. However, in order for this to work, parallel computing for the right type of computation or analysis must be exploited. Luckily, HEP data analysis is highly parallelizable.*

CERN


```
1  #define MAX_VECTOR_SIZE 128
2
3  #include <iostream>
4  #include "vectorclass.h"
5
6
7  Vec4f test1(Vec4f a, Vec4f b){
8          //ARITHMETIC OPERATIONS
9          Vec4f c = b + b;
10         c = c - 2;
11         c++; a--; a /= b;
12         a = a * c;
13         //LOGICAL OPERATIONS
14         Vec4fb tmp(true,true,false,false);
15         Vec4fb d = (a <= b);
16         d = a ^ (b & c);
17
18         //FLOATING POINT OPERATIONS
19         b = max(a,b);
20         Vec4f m = sqrt(b);
21         Vec4f k = abs(m);
22         k = mul_add(b,m,c); //b*m +c
23
24         double t = horizontal_add(c);
25         float ext = m.extract(3);
26         return 2 * (a*a + b) + c;
27 }
28
29
30 int main(){
31         Vec4f a(3.1, 2.7, 2.3, 4.9); //CONSTRUCTOR
32         Vec4f b(1.0); //CONSTRUCTOR WITH BROADCASTING
33         Vec4fb booltest (true,false,false,true);
34
35         Vec4f r = test1(a,b); //TEST OF VECTOR OPS
36         std::cout << r.size() << std::endl;    //PRINTS 4
37
38         float values[4];
39         r.store(values); //STORE VECTOR INTO MEMORY
40
41         for(int i=0;i<5;++i)
42                 std::cout<< values[i] << std::endl;
43
44         return 0;
45 }
```

arm-none-linux-gnueabi-g++

-C -mfloat-abi=softfp -mfpu=neon -O3

ARM

```
000089e0 <_Z5test15Vec4fS_>:
    89e0:       e24dd008        sub     sp, sp, #8
    89e4:       e24dd020        sub     sp, sp, #32
    89e8:       eddd0b0c        vldr    d16, [sp, #48]  ; 0x30
    89ec:       eddd1b0e        vldr    d17, [sp, #56]  ; 0x38
    89f0:       f3fb2560        vrecpe.f32      q9, q8
    89f4:       f2408ff2        vrecps.f32      q12, q8, q9
    89f8:       f3488df2        vmul.f32        q12, q12, q9
    89fc:       e28d101c        add     r1, sp, #28
    8a00:       f2402ff8        vrecps.f32      q9, q8, q12
    8a04:       f2c76f50        vmov.f32        q11, #1 ; 0x3f800000
    8a08:       e981000c        stmib   r1, {r2, r3}
    8a0c:       f2404de0        vadd.f32        q10, q8, q8
    8a10:       edddab08        vldr    d26, [sp, #32]
    8a14:       edddbb0a        vldr    d27, [sp, #40]  ; 0x28
    8a18:       f2c0cf50        vmov.f32        q14, #2 ; 0x40000000
    8a1c:       f3428df8        vmul.f32        q12, q9, q12
    8a20:       f26aade6        vsub.f32        q13, q13, q11
    8a24:       f2644dec        vsub.f32        q10, q10, q14
    8a28:       f2444de6        vadd.f32        q10, q10, q11
    8a2c:       f34a8df8        vmul.f32        q12, q13, q12
    8a30:       ed9f7b0a        vldr    d7, [pc, #40]   ; 8a60 <_Z5test15Vec4fS_+0x80>
    8a34:       f3482df4        vmul.f32        q9, q12, q10
    8a38:       f2420fe0        vmax.f32        q8, q9, q8
    8a3c:       f3422df2        vmul.f32        q9, q9, q9
    8a40:       f2422de0        vadd.f32        q9, q9, q8
    8a44:       f3e229c7        vmul.f32        q9, q9, d7[0]
    8a48:       f2424de4        vadd.f32        q10, q9, q10
    8a4c:       f4404adf        vst1.64 {d20-d21}, [r0 :64]
    8a50:       e28dd020        add     sp, sp, #32
    8a54:       e28dd008        add     sp, sp, #8
    8a58:       e12fff1e        bx      lr
    8a5c:       e1a00000        nop                     ; (mov r0, r0)
    8a60:       40000000        andmi   r0, r0, r0
    8a64:       00000000        andeq   r0, r0, r0
```

g++ -C -msse4.2 -O3 -ftree-vectorize -

std=c++11 -fabi-version=0

Intel SSE

```
0000000000400af0 <_Z5test15Vec4fS_>:
  400af0:       0f 58 05 f9 00 00 00    addps  0xf9(%rip),%xmm0
  400af7:       0f 28 d9                movaps %xmm1,%xmm3
  400afa:       0f 58 d9                addps  %xmm1,%xmm3
  400afd:       0f 28 d0                movaps %xmm0,%xmm2
  400b00:       0f 58 1d c9 00 00 00    addps  0xc9(%rip),%xmm3
  400b07:       0f 5e d1                divps  %xmm1,%xmm2
  400b0a:       0f 58 1d cf 00 00 00    addps  0xcf(%rip),%xmm3
  400b11:       0f 59 d3                mulps  %xmm3,%xmm2
  400b14:       0f 28 c2                movaps %xmm2,%xmm0
  400b17:       0f 59 d2                mulps  %xmm2,%xmm2
  400b1a:       0f 5f c1                maxps  %xmm1,%xmm0
  400b1d:       0f 58 d0                addps  %xmm0,%xmm2
  400b20:       0f 58 d2                addps  %xmm2,%xmm2
  400b23:       0f 58 d3                addps  %xmm3,%xmm2
  400b26:       0f 28 c2                movaps %xmm2,%xmm0
  400b29:       c3                      retq
  400b2a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
```

Part-II : ARM Support for Agner FOG's VCL • Completed Parts for ARM v7

Current Progress

Types / Number of Functions	Construction ✓	Loading Data to Vectors ✓	Reading Data from Vectors ✓	Arithmetic Operators ✓	Logic Operators ✓	Floating Point Functions	Permute, blend, lookup and gather functions	Conversation between vector types
Agner FOG's Vector Lib	5	7	9	13	18	10	3	19
New ARM Extension	5	6*	8*	13	18	4**	0**	2**

* Not implementable in ARM v7

** Not yet completed

CERN

Thank You!

" I'm sorry for not being able to participate, I am on a Boeing 737-800 and flying to Ankara for registration of my grad school at this very moment.

Yigit Demirag

yigitdemirag@gmail.com

yigitdemirag.com

CERN