Vectorization Studies of Random Number Generators on Intel's Haswell Architecture

September 2014

Author:
Yigit Demirag

Supervisors:
Dr. Sandro Wenzel (PH/SFT)
Daniel Funke (PH/SFT)

CERN openlab Summer Student Report 2014

CERN openlab

Project Specification

This project concerns the field of vectorization for Computing in High Energy Physics at CERN, Geneva. This paper summarises the results and progress of vectorizing two newly proposed counter based random number generators on Intel's Haswell Architecture.

Abstract

This project studies SIMD optimizing two different newly proposed random number generators on Intel's Haswell architecture with AVX2 instruction sets. AVX2 instruction set is necessary since many random number generators rely on 64-bit integer multiplication. In first phase, mathematical algorithms behind the random number generators are studied and the places where they can be vectorized are identified. Then all internal data structures of random number generators are transformed from Array of Struct to Struct of Array for better auto-vectorization. To achieve better results intrinsics are used via a high-level C++ wrapping library. In second phase we performed benchmarks and studied the speed up obtained up to 1.57 times for Threefry CBRNG due to vectorization on Haswell.

Table of Contents

1 Introduction 5

2 Vectorization Techniques 6

2.0.1 Finding Blocks to Vectorize 6

2.0.2 Aligning Data 7

2.0.3 Preventing Data Dependencies 7

2.0.4 Avoiding Pointer Aliasing 7

2.0.5 Inlining Function Calls 8

2.0.6 Trying Autovectorization 8

2.0.7 Forcing The Compiler to Autovectorize 8

2.0.8 Performance Improvement 9

3 Benefiting from Auto-vectorization 10

4 Agner Fog's C++ Vector Class Library 11

5 Benchmarking 11

5.1 Evaluation Setup 12

5.1.1 Comparison of AoS,SoA and FOG Implementation on Threefry CBRNG 13

5.1.2 Comparison of AoS,SoA and FOG Implementation on XorShift RNG 14

6 Conclusion and Recommendations 15

1 Introduction

Today's CPU are highly parallel processors with different levels of parallelism. With advanced processor technology, instead of using unique processor per chip, modern devices contains several processing units and allow running several hardware threads simultaneously on the same socket to increase computational power. For each processor generation, number of cores increases, transistor sizes reduce and processors become faster and more comprehensive.

Pseudo Random Number Generators(PRNGs) are widely used at CERN, especially for Monte Carlo Simulations within GEANT4 and ROOT [2]. To scale massively parallel high-performance computation, PRNGs with sequentially independent state transformations are required. Two newly proposed CBRNGs Philox and Threefry, are designed as independent, keyed transformations of counters which produce excellent statistical properties(long period, no discernable structure or correlation)[8]. On the other hand Xorshift RNGs form a class of PRNGs that generate the next number in their sequence by repeatedly taking exclusive or of a number with a bitshifted version of itself[7].

Exploiting vectorization of RNGs can be an advantage to produce more random numbers using less instructions, in particular on the latest generations of CPU, through SSE4.2 to AVX2 instruction sets. This kind of parallel programming is based on Single Input Multiple Data (SIMD). **This work is focused on this optimization area.**

Vectorization is a very similar process to vector operations in mathematics. The programmer can operate sets of data all at once rather than using scalar values. For instance, it is allowed to XOR eight 32-bit integers in a vector with one vector instruction. The results is also saved in the same type of vector register.

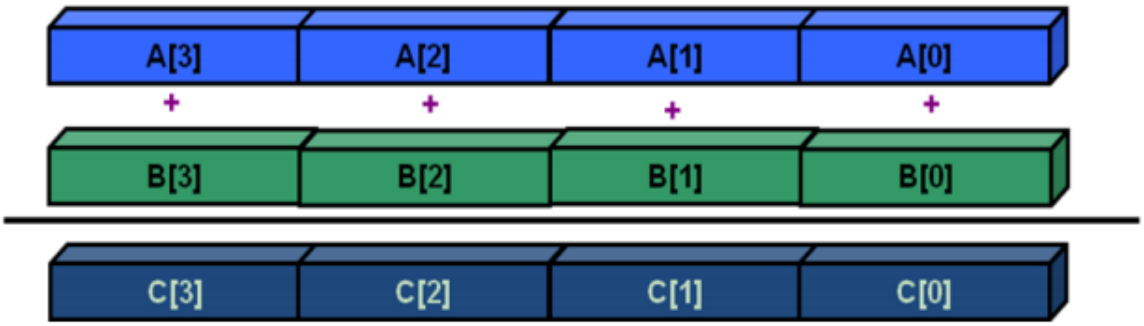

Figure 1: ADD operation in vectorization. With one hardware instruction, it is possible to add four 64-bit integers on Intel's Haswell Architecture

Advanced Vector Extensions (AVX) are extensions to x86 instruction set architecture for microprocessors for Intel and AMD. AVX2, also known as **Intel's Haswell Architecture** extends most integer commands to 512-bit compared to 256-bit vector registers of previous AVX extension. These vector registers are called YMM0-YMM15. In processors with only AVX support, the legacy SSE instructions can be used operating a lower 128-bit of YMM registers (YMM0-YMM7). AVX2 is also the first instruction set supporting three-operand general purpose bit manipulation and multiplication and fused multiply-accumulate(FMA3)[4].

2 Vectorization Techniques

There are several ways to develop vector code with GCC / Intel Compiler, for instance : autovectorization, pragma SIMD statement or built-in function calls called intrinsics. The compiler can help us with vectorizing code in compile time with specific options such as **-vec-report** (in case of Intel Compiler) and can display the diagnostic information reported by the vectorizer after analyzing the vectorizable loops. It also can provide any assumed or proven data dependencies in the loops. To be able to be vectorized, loops must meet following criteria:

Countable loops
: The loop trip must be known at entry to the loop at runtime which means that exiting from the loop does not depend on data.

Single entry and single exit
: There must be no data dependent break in the loop.

Straight-line code
: Since SIMD instructions perform the same operation on data elements from multiple iterations of the original loop, it is not possible for different iterations to have different control flow, i.e., they must not branch. Thus switch statements are not allowed.

No function calls
: There should be no function calls inside the loop that will be vectorized.Two major exceptions are inlined functions and intrinsics math functions.

No data dependency between iterations
: All iterations in the loop has to be completely independent as each SIMD instruction operates on several data elements at once

Contiguous memory access
: In order to have an efficient vectorization, consecutive variables in memory layout have to be loaded directly to vector register with a single vector instruction.

There are some general guidelines for how to write a vectorizable code. Since the order matters, following sections will explain the way we preferred writing a vectorizable code in a structured way with code snippets from different random number generator algorithms we've studied.

2.0.1 Finding Blocks to Vectorize

First, one has to find hot loops/basic blocks to vectorize. For this, we used Intel's VTune™Amplifier and Intel's Software Emulator. Those tools usually give users ability to study the binary code of specific parts of C/C++ code. Therefore one can find which loops are vectorized or which parts of code still possibly can be. Intel's VTune™Amplifier also can provide the information of how much time is elapsed on which function call, while Intel SDE can provide with the numbers of AVX2, AVX1 or SSE4.2 instructions generated in specific function calls. Valgrind, another dynamic analysis tool, is also helpful for analysing the total CPU load on certain parts of code so that looking at conspicuous number of scalar instructions, it is easy to locate blocks that has to be vectorized.

2.0.2 Aligning Data

The next step is to make sure that arrays and structs are aligned. Iterating through multi-dimensional arrays may affect alignment if columns/rows are not a multiple of cache line length. For this kind of problem, solution can be padding or adapting the algorithm. Intel's VTune™can report split loads and and stores with highlighting the application or function call using misaligned data. Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware. Alignment is a property of a memory address, expressed as the numeric address modulo a power of 2 and when it is done, the compiler can create data objects in memory on specific byte boundaries. There are two possible ways to align data :

Creating arrays with a certain byte alignment properties
Alignment attributes can be add when declaring variables to guarantee the're aligned such that **_declspec(align(16, 8))** for Intel Compiler **__attribute__((aligned(16)))** for GCC.

Inserting alignment pragma directives
When the compiler assume data is not aligned, **#pragma vector align** hint can be used to override compiler assumption

Since data alignment depends on processor architecture one should choose 16-byte alignment that facilitates the use of SSE-aligned load instructions when SSE2 platform is targeted. When target is Intel Haswell Architecture, aligning 32-byte boundary gives better results.

Even if compilers are good at automatically handling alignment, sometimes it is hard to check whether they're successful or not. To check this, either we located blocks or the loops causing "vectorization possible but seems inefficient" warning at vector report or we used **objdump** to glanced at assembly and looked for unaligned instructions in tight loops (e.g. mov**u**, vmov**u**). Intel's VTune™Amplifier and Valgrind are also used to visualize CPU cycle waste spent in data access (L1 cache miss, TLB misses, etc)

2.0.3 Preventing Data Dependencies

Threefry, Philox and Xorshift algorithms have no data dependency between the loop iterations and read-after-write (RAW) dependencies.

2.0.4 Avoiding Pointer Aliasing

The **restrict** keyword is used in C99 to assert that the memory referenced by a pointer is not aliased which means that there are no other pointers point that memory location. When **restrict** keyword is used, the compiler will not do any runtime checks for memory aliasing.Also, Intel Compiler interprets **restrict** keyword as it is and **-restrict** option is used while GCC can interpret **__restrict__** and no option is required. We used following macro to avoid compiler dependent problems.

```
#ifdef __INTEL_COMPILER
#define RESTRICTED restrict
#else
#define RESTRICTED __restrict__
#endif
```

Then it is implemented in code as follows :

Table 1: Options that is used for Intel Compiler and GCC for evaluating Intel's different architectures.

Option	Description
-xsse4.2	The compiler may generate instructions from SSE to SSE4.2 vector code
-xCORE-AVX-I	The compiler generates instructions for AVX (256 bits) if the processor supports them.
-xCORE-AVX2	The compiler generates instructions for AVX2 (256 bits), only on Intel's Haswell architecture

```
void __attribute__ ((noinline)) produceArrayofRandomSOAN(int * RESTRICTED counters,
int size, uint64_t * RESTRICTED results){
```

2.0.5 Inlining Function Calls

Through vectorization process, functions which are called for measuring their elapsed execution time, are forced not to be inlined for the sake of uniform measurement. On the other hand, rotation functions in Threefry i.e. *RotL_64()* and rounding function *threefry4x64_R()* and all recursive xor-shift functions are inlined to be able to be vectorized. Since Intel Compiler and GCC take **inline**

keyword differently, we inlined rotational functions as follows:

```
static __inline__ Vec4uq RotL_64(Vec4uq x, unsigned int N)__attribute__((always_inline));
static __inline__ Vec4uq RotL_64(Vec4uq x, unsigned int N)
{
    return (x << (N & 63)) | (x >> ((64-N) & 63));
}
```

During vectorization studies, we observed that Intel Compiler is not able to inline functions from vectori256.h and vectori128.h of Agner FOG's vector class[6]. Therefore we change all **inline** keywords with **inline __attribute__((always_inline))** to force Intel Compiler for inlining.

2.0.6 Trying Autovectorization

During vectorization studies compilation is always done with **-O3** option which provides an aggressive data dependency analysis and loop transformations such as Fusion, Block-Unroll-and-Jam and collapsing IF statements and vectorization is enabled unless otherwise wanted to be examined.

In order to evaluate various Intel microarchitectures with different instruction sets, several compiler options were used. Notice that GCC supports both **-mavx2** and **-march=core-avx2** options and the main difference between them is **-mavx2** supports only for Intel AVX2 intrinsics, built-in functions and code generation[3].

2.0.7 Forcing The Compiler to Autovectorize

The compilers usually have some built-in efficiency heuristics to decide if vectorization is likely to improve performance. **#pragma** statements are used for override those assumptions made by the compiler which prevents vectorization, and declare the compiler that it is safe to ignore such issues.

#pragma vector always statement were used before the loop to ask the compiler to vectorize following loop regardless of the outcome of efficiency analysis.**#pragma ivdep** were used to tell the compiler that any assumed vector dependency is wrong and the loop can be vectorized safely.

```
ks[4][0] =  SKEIN_KS_PARITY64;
ks[4][1] =  SKEIN_KS_PARITY64;
ks[4][2] =  SKEIN_KS_PARITY64;
ks[4][3] =  SKEIN_KS_PARITY64;

#pragma vector always
#pragma ivdep
for (i=0;i < 4; i++){
    for(j=0;j < 4; j++)
    {
        ks[j][i] = k.v[j][i];
        X.v[j][i]  = in.v[j][i];
        ks[4][i] ^= k.v[j][i];
    }
}
```

This code piece from Threefry rounding function were able to be vectorized after two statements.

2.0.8 Performance Improvement

All loops in Threefry CBRNG can be automatically unrolled by both Intel Compiler and GCC compiler. However, we did not observe any automatic unrolling by the compilers at Fog library implementation of xorshift algorithm which caused a performance lose. Therefore unrolling is manually implemented as follows.

The very first version of 64-bit xorshift algorithm with Fog's library implementation:

```
void xorShift64Agner(Vec4uq &x, const int N, Vec4uq &seed, uint64_t * RESTRICTED result)
{
        Vec4uq tmp;
        for(int i=0;i<N/4;i++){
                tmp = (x^(x<<13));
                x = seed;
                seed = (seed^(seed>>35))^(tmp^(tmp>>29));
                seed.store(result+i*4);
        }
}
```

After manual unrolling:

```
void xorShift64Agner(Vec4uq &x, const int N, Vec4uq &seed, uint64_t * RESTRICTED result)
{
        Vec4uq tmp;
        for(int i=0;i<N/4;i+=4){
                tmp = (x^(x<<13));
```

```
                x = seed;
                seed = (seed^(seed>>35))^(tmp^(tmp>>29));
                seed.store(result+i*4);

                tmp = (x^(x<<13));
                x = seed;
                seed = (seed^(seed>>35))^(tmp^(tmp>>29));
                seed.store(result+(i+1)*4);

                tmp = (x^(x<<13));
                x = seed;
                seed = (seed^(seed>>35))^(tmp^(tmp>>29));
                seed.store(result+(i+2)*4);

                tmp = (x^(x<<13));
                x = seed;
                seed = (seed^(seed>>35))^(tmp^(tmp>>29));
                seed.store(result+(i+3)*4);
        }
}
```

3 Benefiting from Auto-vectorization

To improve memory utilization and cache hits, all internal data structure of Threefry and Philox are transformed from Array-of-Structure (AoS) to Structure-of-Array(SoA). The main reason behind using SoA data structure was that AVX2 vector instructions tends to work much better with horizontal memory layouts(SoA). Considering that Intel's Haswell Architecture has L0 and L1 cache lines size of 32 byte each and AVX2 provides vector length of 256-bit, 32byte data alignment with SoA data structure enables to compute 4 at a time without any cache miss and cache pollution.

AoS data structure allows only vectorization for internal calculation that results in generation of 1 random number.

```
struct r123array4x64 {
    uint64_t v[4];
    typedef uint64_t value_type;
    typedef uint64_t* iterator;
    ...
    enum {static_size = 4};
};
```

But SoA data structure allows vectorization not only for 1 random number generation but also 4 random number generation as a stream.

```
struct ALIG32 r123array4x64SOAN {
    uint64_t v[4][4];
```

```
    typedef uint64_t value_type;
    typedef uint64_t* iterator;
    ...
    enum {static_size = 16};
};
```

4 Agner Fog's C++ Vector Class Library

Agner Fog's vector class is a free software library assisting with the vectorization of C++ code[6]. This library is a high level API to use specific instruction sets from SSE to AVX2, and implemented using *instrinsics*. The interface of Fog's vector class ease efficient development of vectorized algorithms by abstracting the low level programming. Following source code piece demonstrates how addition operator is defined in Fog's vector class :

```
// vector operator + : add element by element
static inline Vec32c operator + (Vec32c const & a, Vec32c const & b) {
    return _mm256_add_epi8(a, b);
}
```

The a and b vector objects are the addresses of vectors registers storing 8 32bit unsigned integers.And **_mm256_add_epi8(a, b);** adds packed 8-bit integers and store result variable in the same kind of vector register. During compile time, library decides which instruction sets to support. It is possible to work with SSE2,SSE4,AVX or AVX2.

5 Benchmarking

To properly measure the performance of algorithms, we created a test file including 3 functions for 64-bit Threefry to compare standard, auto-vectorized and Fog's library implementations; and 4 functions for each 32-bit xorshift and 64-bit xorshift algorithms to compare normal, auto-vectorized and Fog's library options specified for AVX2 and SSE4.2 microarchitectures.

Test is performed with generating 80 million random numbers. 3 different Threefry implementations are tested with same key and counter pairs and generated exactly same random numbers. In case of Xorshift algorithms four 32-bit and four 64-bit algorithms took same two 32-bit words and two 64-bit words respectively and generated same random numbers.

Benchmarking is done via Timer.h, a free software library providing precise time measurement. Timer.h defines **Timer** class, which implements a simple start/stop timer that can tell you how much time has elapsed since it was called.

In benchmarking, performance of algorithms with no-specific-platform, SSE4.2, AVX1 and AVX2 specified instruction sets are compared with and without auto-vectorization generated by Intel Compiler and GCC on an AVX2 machine. .

Table 2: The list of executables generated in the script.

Auto Vect.	Platform	Command
ICC		
No	no-specific	-O3 -no-vec newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
Yes	no-specific	-O3 newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
No	SSE4.2	-O3 newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
Yes	SSE4.2	-xSSE4.2 -O3 newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
No	AVX-1	-xCORE-AVX-I -O3 -no-vec newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
Yes	AVX-1	-xCORE-AVX-I -O3 newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
No	AVX-2	-xCORE-AVX2 -O3 -no-vec newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
Yes	AVX-2	-xCORE-AVX2 -O3 newTest.cpp xorShift.cpp -inline-level=1 -std=c++11 -restrict
GCC		
No	no-specific	-O3 -fno-tree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
Yes	no-specific	-O3 -ftree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
No	SSE4.2	-msse4.2 -O3 -fno-tree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
Yes	SSE4.2	-msse4.2 -O3 -ftree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
No	AVX-1	-march=core-avx-i -O3 -fno-tree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
Yes	AVX-1	-march=core-avx-i -O3 -ftree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
No	AVX-2	-march=core-avx2 -O3 -fno-tree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
Yes	AVX-2	-march=core-avx2 -O3 -ftree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
No	AVX-2	-mavx2 -O3 -fno-tree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0
Yes	AVX-2	-mavx2 -O3 -ftree-vectorize newTest.cpp xorShift.cpp -std=c++11 -fabi-version=0

5.1 Evaluation Setup

3 Threefry, 4 32-bit xorshift and 4 64-bit xorshift functions are called from a test file and the elapsed time is written to standard error stream, **std::cerr** such that :

```
0.3545 1
0.3312 1.07
0.2495 1.42
0.2282 1.55
0.09396 3.77
0.04244 8.35
0.04819 7.36
0.1888 1.88
0.09064 3.91
0.08542 4.15
0.1261 2.81
```

where first column is time elapsed in seconds by AoS, SoA and Fog implementation of Threefry; standard, auto-vectorization, AVX2 Fog and SSE4.2 Fog implementation of 32-bit and 64-bit xorshift algorithm. The second column displays the relative speed up of algorithms compared to first one, which is AoS implementation of Threefry. Second column is not used in benchmarks.

For each compiler and options given below, different executables are generated using a bash script.

Then all executables are run to get elapsed times of different algorithms. At this point, **taskset -c 0** option is used before executing files to run executable on only one processor and CPU frequency is

set to 3.0 GHz by CPU governer to avoid fluctuations in frequency during calculations.

```
i=100;
for f in $(ls ex/*_80000000_*); do
   i=$(( $i + 1 ));
  taskset -c 0  ./$f 1> ex/data/${i}_output;
done
```

Finally, performance histograms are generated via a Python script. In plots, performance of different options are normalized with performance of non-vectorized, no-specific platform option for more easier comparison.

5.1.1 Comparison of AoS,SoA and FOG Implementation on Threefry CBRNG

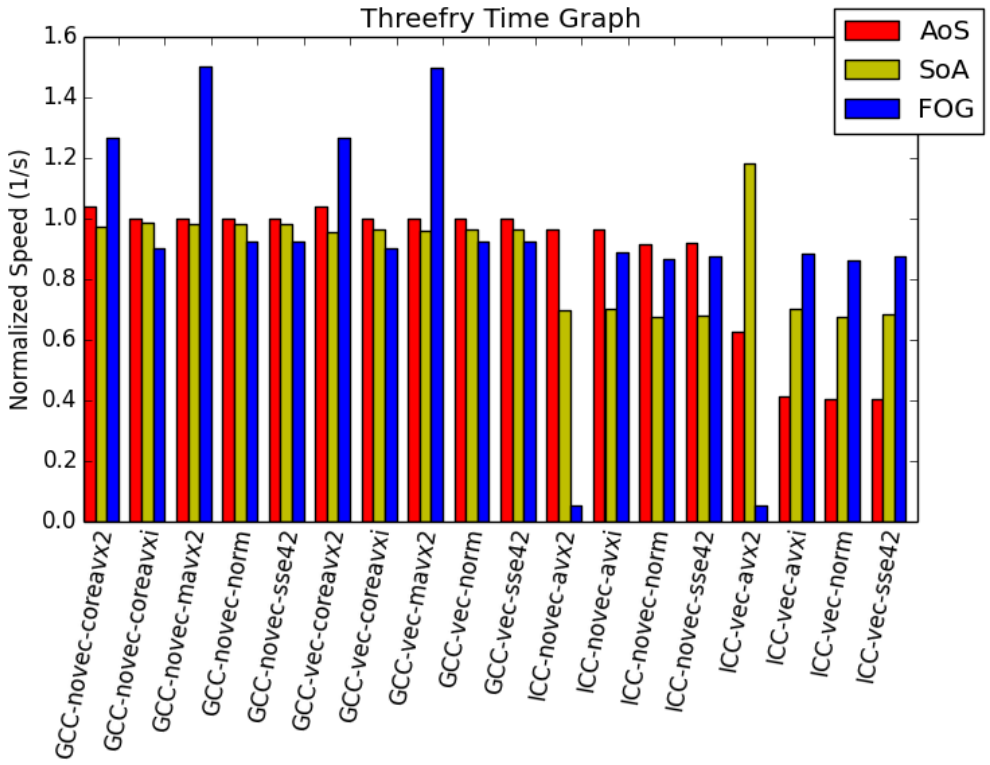

Figure 2: Vectorization performance of Threefry on Different Architecture and Compiler Options.

For Threefry CBRNG algorithm, only Fog library implementation with AVX2 specific instructions that is generated by GCC and auto-vectorization by Intel Compiler with **-xCORE-AVX2** options performed a clear speed up. The best performance with 1.57 times speed up is achieved by implementing Fog's vector class when compiled with GCC using **-mavx2** option. Also it seems that Intel Compiler's effort to optimize the code for AVX2, interfered with FOG library and resulted in performance lost, up to 90%. And finally, plot shows that to vectorize Threefry, AVX2 is preferable to SSE4.2 in 64-bit integer operations when intrinsics is used via Fog's vector class instead of auto-vectorization.

5.1.2 Comparison of AoS,SoA and FOG Implementation on XorShift RNG

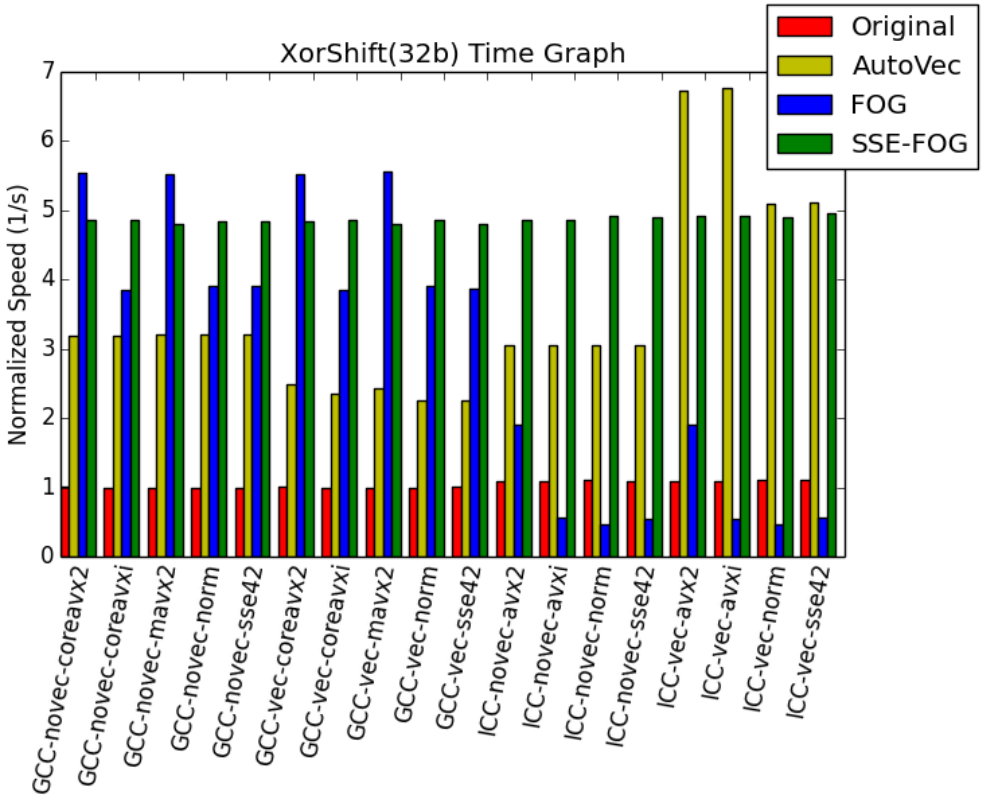

Figure 3: Vectorization performance of 32-bit xorshift on Different Architecture and Compiler Options.

For 32-bit xorshift algorithm, data structure aiming auto-vectorization performed a clear speed up independent whether auto-vectorization is enabled or not. Also it appears that GCC with **-no-vec** option performed better performance on that data structure. The reason of this interesting event could be that memory operations in horizontal memory layout is faster than vertical layout because of higher chance of hitting cache. On the other hand Intel Compiler did a better job and vectorized the code and resulted in 6.8 times speed up. Also it is clear that to speed up 32-bit xorshift algorithm, AVX2 platform is more preferable than SSE4.2 when internal data structure is transformed for better memory access and Intel Compiler's auto-vectorization is enabled.

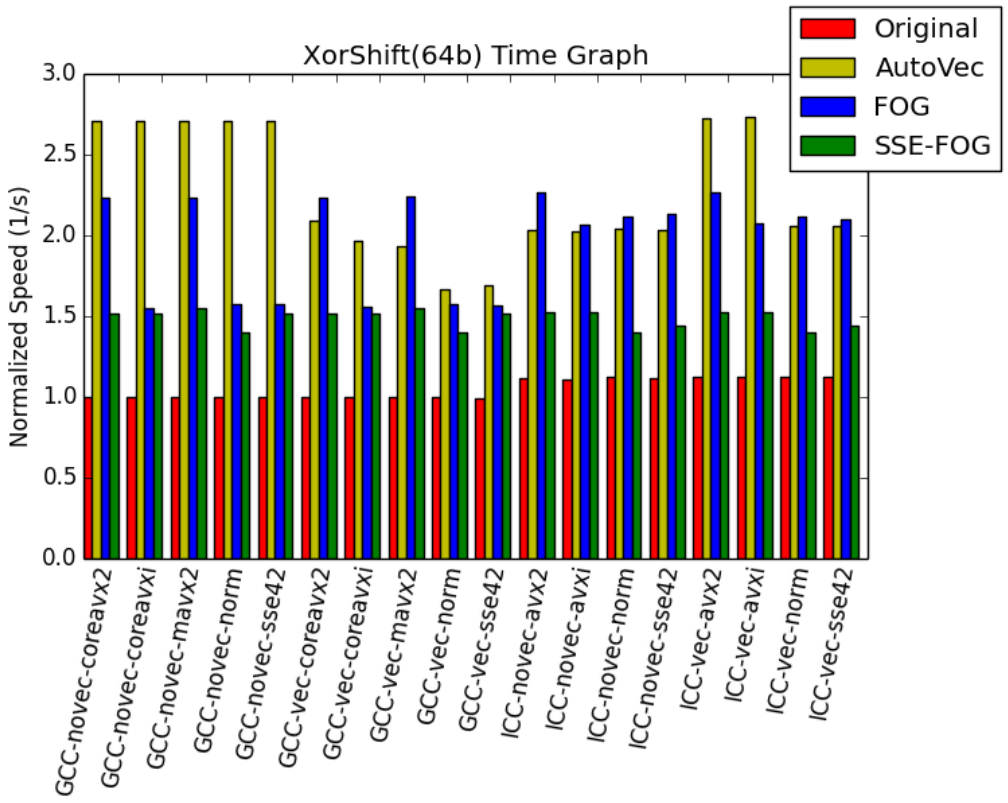

Figure 4: Vectorization performance of 64-bit xorshift on Different Architecture and Compiler Options.

For 64-bit xorshift algorithm, data structure aiming auto-vectorization and Fog's vector class library performed a clear speed up compared to standard version. Also it appears that GCC with **-no-vec** option performed better performance on that data structure as in the case of 32-bit xorshift algorithm. Since SSE version of Fog's vector class implementation requires to work with vector components of 32-bit, we couldn't observe a clear performance compared to Fog's library implementation working with vector components of 64-bit. The best performance is observed with data-structure aiming auto-vectorization, up to 2.7 times. However, no particular gain of AVX2 platform over SSE4.2 is observed.

6 Conclusion and Recommendations

Thanks to Intel' Haswell Architecture, it is possible to work with wider vector registers. To get good performance out of AVX2, applications need to take advantage of vectorization that allows integer operations of four numbers of 64-bit with a single instruction.

To benefit from vectorization and to examine the performance gain of AVX2 over SSE4.2, newly proposed Threefry CBRNG and xorshift algorithm as a case study, are studied. Although the vectorization study is not completed, 1.57 times speed up for Threefry and 2.7 times speed up for xorshift are observed due to vectorization by Fog's vector class library and auto-vectorization by Intel Compiler respectively.

As a recommendation, the further research has to be done on caching and data access issues of Threefry, precisely on allocations of counter and key structs. Considering vectorization of Philox CBRNG, widening multiplication that prevents flat-out vectorization should be studied, on the other hand, Haswell's *MULX* instruction which has more flexible register use may help to solve the problem.

References

[1] Cerion Armour-Brown. Valgrind 3.10.0. http://valgrind.org/downloads/, September 2014.

[2] Rene Brun. How to use root with monte carlo programs. Technical report, CERN, 2014.

[3] Martyn Corden. Intel® compiler options for intel® sse and intel® avx generation (sse2, sse3, ssse3, atom_ssse3, sse4.1, sse4.2, atom_sse4.2, avx, avx2) and processor-specific optimizations. https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and August 2012.

[4] Intel Corporation. New microarchitecture for 4th gen intel® core™ processor platforms. Technical report, Intel Corporation, 2013.

[5] Intel Corporation. Intel® vtune™ amplifier 2015. https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2015.

[6] Agner Fog. Vcl c++ vector class library. Technical report, 2012-2014.

[7] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 7 2003.

[8] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. 2011.

[9] Ady Tal. Intel® software development emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator, June 2012.