Creating a compiler optimized inlineable implementation of INTEL SVML SIMD intrinsics

Accepted 23rd May, 2018

ABSTRACT

Single Input Multiple Data (SIMD) provides data parallelism execution through implemented SIMD instructions and registers. Most mainstream computers architectures have been enhanced to provide data parallelism through SIMD extensions. These advances in parallel hardware have not been accompanied by the necessary software libraries granting programmers a set of functions that could work across all major compilers adding a 4x, 8x or 16x performance increase as compared to standard SISD. Intel's SVML library offers SIMD implementation of Mathematics and Scientific functions that have never been ported to work outside of Intel's own compiler. An Open Source inlineable implementation would increase performance and portability. This paper illustrates the development of an alternative compiler neutral implementation of Intel's SVML library.

Keywords: Data Parallelism, SIMD Intrinsics, SVML, C++ Math Library, Computer Simulation, Scientific Mathematics Library.

INTRODUCTION

Data parallelism occurs when one or more operations are repeatedly applied to several values. Generally, programming languages sequentially programme scalar values. Single Input Multiple Data (SIMD) compiler intrinsics allow operations to be grouped together leading to improved transistors per Floating Point Operations reducing power usage and overall modern SIMD capable of CPUs. SIMD has some limitation because you cannot easily vectorize different functions with lots of statements. These limitations define SIMD Intrinsic programming to be almost a new programming paradigm within languages like C or C++.

In most modern compilers the work of optimization can generally outperform many programmer tricks to generate optimal output. In the case of SIMD, the compiler is smart enough to transform Vector/Arrays into optimized SIMD operations. The idea is simple, instead of computing one value of an array at a time your compute identical operation on all values is able to fit into SIMD register. All data must be subject to the same processing. With the addition of compiler pragmas such as Intel® Cilk™ Plus (https://www.cilkplus.org/), a programmer is able o force the compiler to work harder on specific parts of code to optimize data-parallelism. However, this is not always the case since the compiler cannot re-order all scalar operation to parallel operations. It is possible to do conditional processing within SIMD; however, it requires some tricks since the compiler does not understand how to implement from traditional scalar code.

In the case where automatic vectorization is not an option there needs to be a library of fictions a programmer can target. This library should be portable enough to be able to port code written for this library to be able to be compiled by the main C/C++ compilers that support Intel Intrinsics such as GCC, CLANG and MSVC. Many vendors have specified an API for Intrinsic such as Intel® short vector math library (SVML) (https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SVML), a software standard provided by Intel® C++ Compiler (https://software.intel.com/en-us/node/524289).

SVML functions can be broken down into five (5) main
categories: Distribution, Division, Exponentiation, Round, Trigonometry and subcategories of __m128, __m128d, __m128i, __m256, __m256d, __m256i, __m512, __m512d, __m512i corresponding to vector data types. When implementing SVML functions, the functions can be approximated using Approximation of Taylor series expansion in combination with bitwise manipulation of IEEE745 floating point representation (Malossi et al., 2015; Nyland, 2004; Kretz, 2015). Reference implementation of scalar version can be translated into SIMD version from Cephes library (http://www.netlib.org/cephes). The output of the replacement functions can be tested and compared to output form SVML library provided in Intel® C++ Compiler (https://software.intel.com/en-us/node/524289).

BITWISE SELECTION

Due to the nature of SIMD functions operating entirely bitwise, sequential functionality must be considered and reworked for sequential implementation of some non-sequential/bitwise algorithms. This includes conditions where an algorithm can jump from one point of the stackframe to a point in order to compute one value that would be common in SISD algorithms (Figure 1).

Bitwise selection occurs when vector bitmask is use to select values within the vector. Vector bitmask is produced when SIMD comparative operation is used to compare two (2) vectors resulting in mask of all 1 for true or mask of all 0 for false. Each mask corresponds to value held in vector position whose size is allocated to each data type byte alignment (Figures 2 to 5).

It is important to understand bitwise selection in SSE SIMD operations since they will take the pace of jumping/branching based on condition in most arithmetic operation. These bitwise operations become fundamental in the design and analysis of SIMD optimized algorithms. Since all data operation are identical across SIMD vector code segments must be broken up based on condition and properly placed into the correct vector portion without resorting to scalar operations to do so. It is important to note that SSE4.1 blendv bitwise selection under the hood is performing the operation of a bitwise OR the result of a (~mask and first_input_value) and the result of the (mask and second_input_value). This function is completely bitwise in nature and depends on binary vector representation of TRUE and FALSE to allow the pass of values within each vector position within the SIMD register (Figure 6).

Once a programmer understand that bitwise selection masks are used in place of conditional statements the porting of existing C/C++ library becomes easier. The programmer would still have to account for inefficiency of computing all possible branches and corner cases which would later merge into the resulting SIMD vector. With this in mind unnecessary corner cases and redundancy within
\_m128 \_mm\_radian\_asin\_ps\( (\_m128 \ a) \)
{
    \_m128 y, n, p;
    p = \_mm\_cmpge\_ps(a, \_m128\_1);
    // if(a >= 1) Generates Mask 0xffffffff -OR- 0x0
    n = \_mm\_cmple\_ps(a, \_m128\_minus\_1);
    // if(a <= -1) Generates Mask 0xffffffff -OR- 0x0
    // 0xffffffff is TRUE
    // 0x0 is FALSE
    y = \_mm\_asin\_ps(a);
    y = \_mm\_blendv\_ps(y, \_m128\_PIO2F, p);
    // replace value for >= 1 in var y based on 0xffffffff mask
    y = \_mm\_blendv\_ps(y, \_m128\_minus\_PIO2F, n);
    // replace value for <= -1 in var y based on 0xffffffff mask
    return y;
}

Figure 2: SSE SIMD branching example.

\_mm\_radian\_asin\_ps\( (\_m128)\):
push rsi
movaps xmm9, xmm0
movups xmm8, XMMWORD PTR \_m128\_1[rip]
cmpleps xmm8, xmm0
cmpleps xmm9, XMMWORD PTR \_m128\_minus\_1[rip]
call \_svm\_asin4
movaps xmm1, xmm0
movaps xmm0, xmm8
blendvps xmm1, XMMWORD PTR \_m128\_PIO2F[rip], xmm0
movaps xmm0, xmm9
blendvps xmm1, XMMWORD PTR \_m128\_minus\_PIO2F[rip], xmm0
movaps xmm0, xmm1
pop rcx
ret

Figure 3: SSE SIMD branching assembly language output (Intel C++ Compiler).

float radian\_asin(float a) {
      if(a >= 1) { return PIO2F; }  
      else if(a <= -1) { return -PIO2F; }  
      else { return asin(a); }  
}

Figure 4: Standard branching example.

an algorithm should be evaluated as critical or non-critical to compute the final resulting value (Figure 7).
**radian_asin(float):**

```c
push rsi
comiss xmm0, DWORD PTR .L_2il0floatpacket.3[rip]
jae ..B1.6
movss xmm1, DWORD PTR .L_2il0floatpacket.1[rip]
comiss xmm1, xmm0
jb ..B1.4
movss xmm0, DWORD PTR .L_2il0floatpacket.2[rip]
pop rcx
ret
..B1.4:
call asinf
pop rcx
ret
..B1.6:
movss xmm0, DWORD PTR .L_2il0floatpacket.0[rip]
pop rcx
ret
```

Figure 5: Standard branching example assembly language output (Intel C++ Compiler).

```c
__m128i _mm_blendv_si128 (__m128i x, __m128i y, __m128i mask) {
    // Replace bit in x with bit in y when matching bit in mask is set:
    return _mm_or_si128(_mm_andnot_si128(mask, x), _mm_and_si128(mask, y));
}
```

Figure 6: Bitwise selection example code.

```c
__m128 _mm_fmadd_ps(__m128 a, __m128 b, __m128 c)
// Latency ~6

__m128 _mm_add_ps ( _mm_mul_ps (__m128 a, __m128 b), __m128 c)
// Latency ~10
```

Figure 7: FMA vs. SSE CPU Cycle latency.

**FMA INSTRUCTION SET**

The FMA instruction set is an extension to the 128-bit and 256-bit streaming SIMD extension instructions in the Intel Compatible microprocessor instruction set to perform fused multiply-add (FMA). There are two variants: FMA4 is supported in AMD processors starting with the Bulldozer (2011) architecture. FMA4 was realized in hardware before FMA3 was realized in hardware before FMA3 (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf). Ultimately, FMA4 (https://support.amd.com/TechDocs/43479.pdf) was dropped in favor of the superior Intel FMA3 hardware implementation.

FMA3 is supported in AMD processors since processors in 2014. Its goal is to remove CPU cycles by combining multiplication and addition into one instruction. This would reduce the total execution time of an algorithm.
__m128_sign_mask = (0x80000000)

_mm_preservesignbit_ps(const __m128 a) {
    return _mm_and_ps(a, __m128_sign_mask);
}

_mm_effectsignbit_ps(const __m128 a, const __m128 b) {
    return _mm_xor_ps(a, b);
}

Figure 8: Sign preservation functions.

Figure 9: 32-bit IEEE-745 floating point number.

Figure 10: 64-bit IEEE-745 floating point number.

IMPLEMENTATION

SVML compatible library starts with the most commonly called functions falling under the family of libC math functions. Most implementations are portable form__m128 to __m128d, __m128 to __m256/__m512 and __m128d to __m256d/__m512d with minor changes to numerical constants and corresponding intrinsic for specific vector datatype. LibC Math functions are able to be adapted to SSE/AVX family of functions removing corner cases and optimizing the use of native SSE, AVX and FMA extensions to remove unnecessary CPU cycles for SIMD implementation of math functions. Implementation of Rounding intrinsics are mapped to existing SSE intrinsics and Integer Division are unpacked into scalar operations and repacked into vectors and do not need implantation.

Sign preservation

To handle conditional branching of preserving sign we must first implement a vector wide sign bit preservation functions returns a bitmask of the preserved sign mask (Figure 8). This will make any porting of standard C/C++ code to using Intel SIMD intrinsic much easier and more efficient at run time.

Exponentiation

This begins with the __m128 __mm_log_ps(__m128 a) function. It is done through shifting to the right the Mantissa bitwise to extract the exponent field and the Inverse Mantissa Mask (~0x7F80000000000000) to extract the fraction field within the IEEE745 floating point representation (Figure 9).

Once exponent bits are extracted the remaining approximation is made by approximation of Taylor series expansions pre-computed as constants set as the __m128, __m256 or __m512 SIMD vector representation.

64-bit versions of the Inverse Mantissa Mask remains the same, however, the Mantissa __m128i_i32_0x7f representing 127 or 2^8-1 representing exponent bit turns to__m128i_i64_0x400 representing 1024 or 2^11-1 for the 64-bit representation of IEEE-745 floating point representation (Figure 10). The case of __mm_cvtepi32_ps changed to __mm_cvtepi64_pd for 64-bit version that exist inside the AVX-512 [https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX_512] standard though the functions inside the SSE family is missing. These missing functions are able to be backported using SSE (Figure 11).

__m128d_Int64ToDoubleMagic numerical contrast represented by (0x4337FFFFE5B60600) or
\`
\_m128d _mm_cvtepi64_pd(__m128i i64 a) {
    a = __mm_add_epi64(a, __mm_castpd_si128(__m128d_Int64ToDoubleMagic));
    return __mm_sub_pd(__mm_castsi128_pd(a), __m128d_Int64ToDoubleMagic);
}
\`

\`
\_m128i _mm_cvtpd_epi64(__m128d a) {
    a = __mm_add_pd(a, __m128d_Int64ToDoubleMagic);
    return __mm_sub_epi64(__mm_castpd_si128(a), __m128d_Int64ToDoubleMagic);
}
\`

Figure 11: Convert 64-bit integers in to a 64-bit floating point (AVX-512 Backport).

\`
__m128 _mm_abs_ps(const __m128 a) {
    return _mm_andnot_ps(_mm_set1_ps(-0.0F), a);
}
\`

Figure 12: SSE Absolute value.

- In input range is limited using \_mm_min_ps (88.37626647949F) as the high limit and \_mm_max_ps (-88.37626647949F)
- \( ex = x \cdot \log_2(e) + \frac{1}{2} \) Where \( \log_2(e) \) is pre-computed as (1.44269504088896341F)
- \( ex = \text{floor}(ex) \)
- \( y = x \cdot K_d(xf) \), with \( K_d(xf) = a^n x_f^n + b^n x_f^{n-1} + c x_f^{n-2} + ... \)
- \( y = y \cdot x_f^2 + x + 1 \)
- value as the approximated exponential \( e^x = y \) (IEEE745 Exponent filed and the fraction field of variable ex)

Figure 13: Implementation of exp.

(6.755399e+15) special value is used to convert to and from 64-bit and double. Its range is from -\(2^{51}\) to \(2^{51}\) precision vs. native AVX-512 range from -\(2^{63}\) to \(2^{63}\). This range reduction does not have any significant impact however, it adds a few CPU cycles by comparison to native AVX-512 implementation. In the case of the implementation for \_m128 _mm_log2_ps(__m128 a) and \_m128 _mm_log10_ps(__m128 a) the ineligible intrinsic of \_mm_log_ps is multiplied by numerical constants. In the case of \_mm_log2_ps log2(exp(1)) it can be pre-computed into constant (1.44269504088896341F) and then multiplied by log(x) to produce the correct output. In the case of \_mm_log10_ps log10(exp(1)) constant (0.4342944819032518F) it can be multiplied by \_mm_log_ps to produce the correct output. This method is the easiest to implement and does not add to many CPU cycles to the new functions since values are pre-computed and multiplied to an intrinsic called once. For the implementation of \_m128 _mm_logb_ps(__m128 a) the binary logarithm implementation takes the absolute value of the input of \_mm_log2_ps followed by passing the output to built in SSE \_mm_floor_ps functions of the resulting value. The absolute value function just strips the sign bit example of absolute value implementation missing from the SSE family (Figure 12).

For the implementation of \_m128 _mm_exp_ps(__m128 a) it takes a similar approach to the logarithm functions. Implementation is modified from Cephes where approximation is calculated from constants extracting exponent bit. It returns \(e (2.71828...)\) raised to the x power. Unfortunately, Intel never implemented a SSE/AVX equivalent to X87/NPX instruction F2XM1 (https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf) which calculates \(2x-1\) so an approximation is necessary for implementation (Figure 13).

In the 64-bit implementation it is the same as 32-bit implementation, however, the output of previous
implementation of exp is multiplied $\frac{1}{2}$ to account for doubling of precision. For the implementation of _m128 _mm_exp10_ps(_m128 a) we can use the mathematical rule of exp10(x) = exp(x * log(10)) where log(10) can be pre-computed into constant (2.30258509299404568402F) and the input value multiplied and the result is put into exp_mm_exp_ps(_m128 a) to produce the correct output. This would work the same way for 64-bit version. In the case of _m128 _mm_exp2_ps(_m128 a) we could use the same rule exp2(x) = exp(x * log(2)). It is important to mention that it would not be appropriate to only rely on bitwise shifting in this implementation even though it results may in theory be equivalent to $(2^x)$, however, it will not work in practice. In the case of the implementation of _m128 _mm_pow_ps(_m128 a, _m128 b) the input of base is passed into _mm_abs_ps whose result is passed into _mm_log_ps which is then multiplied by floating point representation exponent with the result passed into _mm_exp_ps. This implementation is efficient enough even though it inline all of _mm_log_ps and _mm_exp_ps. This implementation appears likely to be slower than Intel’s SVML and can be further optimized, however, upon testing it is comparable to Intel’s implementation. An alternate version of power to calculate root can be implemented.

The sign is persevered with _mm_preservesignbit_ps and the final output use the _mm_effectsignbit_ps to return the sign back to the final output. For the reciprocal exponent the _mm_rcp_ps intrinsic which is 12-bit of accuracy or maximum relative error for this approximation is less than $1.5^2^{12}$. This level of precision is sufficient. For 64-bit version, _mm_rcp14_ps intrinsic introduced in AVX-512 is backported. The AVX-512 version uses 14-bit of precision or maximum relative error for this approximation that is less than $2^{14}$. The back ported version uses the full range. An inverse version of the nth root function can then call the reciprocal intrinsic which then can be used to create a _mm_cbrt_ps and _mm_inv_cbrt_ps for 32-bit versions. In addition to _mm_cbrt_pd and _mm_inv_cbrt_pd for 64-bit versions, this would be done by passing a constant of 3 into the exponent parameter.

In the cases of _mm_sqrt_ps and _mm_inv_sqrt_ps they are just alternate names for _mm_sqrt_ps and _mm_inv_sqrt_ps. Finally, _mm_exp1_ps just calls _mm_exp_ps and then subtract 1 from its result, while _mm_log1p_ps adds 1 to the input parameter. With exponentiation library finished trigonometry and distribution portions of SVML are implemented.

### Trigonometry

From the trigonometry libraries we can use exponentiation library to implement hyperbolic trigonometry functions. In the implementation of _m128 _mm_sinh_ps(_m128 a) value of exp(x) can be processed held in a variable and the value of exp(-x) translated to reciprocal of exp(x) using a built-in SSE reciprocal function. This eliminates unnecessary inline of 2 different _mm_exp_ps. This reduces the amount of CPU cycles with no significant loss of accuracy. The value of exp(x) and reciprocal exp(x) are then subtracted and thereafter, divided by a constant of 2. This method of computing sinh(x) only adds 2 additional operations to _mm_exp_ps. In the implementation of _m128 _mm_cosh_ps(_m128 a) it follows the same process as _mm_sinh_ps; however, it replaces the subtraction operation with addition resulting in the same level of efficiency. For the implementation of _m128 _mm_tanh_ps(_m128 a) value of exp(2x) can be processed held in a variable while the value is subtracted by a constant of 1 for the numerator. The denominator is computed by variable added to a constant of 1. The resulting numerator and denominator are then divided. This method of computing tanh(x) only adds 3 additional operations to _mm_exp_ps. For the implementation of _m128 _mm_asinh_ps(_m128 a) the value of x^2+1 will be computed using FMA _mm_fmadd_ps(x, x, _m128_1), thereafter, the result will be passed into built-in _mm_sqrt_ps and then added to input and finally passed to our exponentiation function _mm_log_ps. This implementation only uses 3 additional operations to the inline _mm_log_ps function. The implementation of _m128 _mm_acosh_ps(_m128 a) is the same as _mm_asinh_ps, however, the value of x^2+1 replaced by x^2−1 will be computed using FMA _mm_fmsub_ps(x, x, _m128_1) resulting in the same level of efficiency. When evaluating the implementation of _m128 _mm_atanh_ps(_m128 a) (1 + x / 1 - x) passed to our exponentiation function _mm_log_ps is multiplied by a static constant of $\frac{1}{2}$. This implementation only uses 4 additional operations to the inline _mm_log_ps function. For the implementation of _m128 _mm_hypot_ps(_m128 a, _m128 b) using combination of built-in SSE sqrt() in combination with FMA and SSE is multiplied _mm_fmadd_ps(a, a, _mm_mul_ps(b, b)) giving the best performance. The implementation of _m128 _mm_sin_ps(_m128 a) is a direct port of Cephes (http://www.netlib.org/cephes) (Figure 14).

The implementation of _m128 _mm_cos_ps(_m128 a) is a direct port of Cephes (http://www.netlib.org/cephes) (Figure 15). The _m128 _mm_sincos_ps(_m128 *mem_addr, _m128 a) implementation combines sin() and cos() functions into a single function combining redundant operation that are computed using the same numerical constants. The implementation of _m128 _mm_tan_ps(_m128 a) is a direct port of Cephes (http://www.netlib.org/cephes) (Figure 16).

For implementations of _m128 _mm_sind_ps(_m128 a), _m128 _mm_cosd_ps(_m128 a), and _m128 _mm_tand_ps(_m128 a) the input of _mm_sin_ps, _mm_cos_ps and _mm_tan_ps is multiplied by a static constant of $\pi / 180$. The implementation of _m128 _mm_asin_ps(_m128 a) is also a port of Cephes.
preserve the sign and take the absolute value of the input as variable (a)
input ins multiplied by constant of 4/PI and converted to integer stored as variable i0
i0 = (i0 + 1) + -1
y = conversion of i0 to float
i1 = i0+4 and its result is shifted left 29 places shifting in zeros
i0 = (i0 + 2) and is compared == to 0 where i0 holds the bitwise selection mask and is cast (Not covered) back into float. (Form now on float casted i0 will be known as polymask and i1 is signmask)
the new sign come as a result of xor of signswap by the polymask
a = ((y * -3.7749497744594108) + ((y * -2.4187564849853515625) + ((y * -0.78515625) * a)))
y = a(2/2 + (((((4.43315171809948 * a') - 1.38873162549376) * a') + 4.16664568298287) * a' + a' + a' + a') + 1
y2 = ((((((1.951295981 * a') + 8.3321608736) * a') + 1.666654611E) * a') + a ) + a
select from y and y2 using polymask and return sign to result

Figure 14: Implementation of circular sin.

preserve the sign and take the absolute value of the input as variable (a)
input ins multiplied by constant of 4/PI and converted to integer stored as variable i0
i0 = (i0 + 1) + -1
y = conversion of i0 to float
i1 = i0+4 and its result is shifted left 29 places shifting in zeros
i0 = (i0 + 2) and is compared == 0 where i0 holds the bitwise selection mask and is cast (Not covered) back into float (form now on float casted i0 will be known as polymask and i1 is signmask)
a = ((y * -3.7749497744594108) + ((y * -2.4187564849853515625) + ((y * -0.78515625) + a)))
y = a(2/2 + (((((4.43315171809948 * a') - 1.38873162549376) * a') + 4.16664568298287) * a' + a' + a' + a') + 1
y2 = ((((((1.951295981 * a') + 8.3321608736) * a') + 1.666654611E) * a') + a ) + a
select from y and y2 using polymask and return sign to result

Figure 15: Implementation of circular cos.

preserve the sign and take the absolute value of the input as variable (a)
input ins multiplied by constant of 4/PI and converted to integer stored as variable i0
y = the conversion back from i0 (effectively truncating)
i0 = i0+1 and compared == 1 where i0 becomes comparative mask
i0 = i0 + i1&1
y = y * (((y convert to float) & 1)
z = (y * -3.7749497744594108) + ((y * -2.4187564849853515625) + (y * -0.78515625 + original input )))
y = (((((((9.38540185545 * z') + 3.1199232697) * z') + 2.44301354525) * z' + 5.34121807065) * z' + 1.33387994085) * z') + 3.33331568548) * z') + z)
z' = z/y
i1 = i0 + 2 and its result is compared == 2 making i1 a selection mask
y and z are selected based on i1 and sign is return to resulting output

Figure 16: Implementation of circular tan.

 preserve the sign and take the absolute value of the input as variable (a)
a is compared > 1/2 producing comparative mask
z = (1-a)/2
w = sqrt(z)
a = selection of a and w using comparative mask
z = selection of a' and z comparative mask
y = ((((((((z * 2.4163199048) + 2.418311049) * z) + 4.5470025998) * z) + 7.495302686) * z) + 1.6666752422) * z') + a) + a
z = pi/2 - (y + y)
result is selection between y and z using comparative mask and sign is return to result

Figure 17: Implementation of circular arcsin

(ftp://www.netlib.org/cephes) (Figure 17).
The implementation of _m128_mm_acos_ps(_m128 a) does not call _mm_acos_ps since that would have to be inlined twice creating a performance hit. Implantation used is based around the SunPro/Sun Microsystems version modified by Ian Lance Taylor of Cygnus adopted into many versions of libc such as NewLib (http://sources.redhat.com/newlib) (Figure 18).
In the implementation of _m128_mm_atan_ps(_m128 a) there are ports from Cephes Inverse circular tangent (Figure 19). Our _m128_mm_atan2_ps(_m128 a, _m128 b) implementation calling _mm_atan_ps (a/b) checks for all relevant corner cases. Implementing _mm_atan_ps before _mm_atan2_ps instead of using _mm_atan2_ps(a, _m128 b) to implement _mm_atan_ps is generally faster avoiding computing unnecessary corner cases inherited from _mm_atan2_ps. Total corner case checking adds 11 extra instructions from _mm_atan_ps
• input is variable of a
• gt = comparison of > a and 1/2
• lt = comparison of < a and -1
• nlt = xor(a, 0xFFFFFFFF)
• mt = ~lt & ngt
• z = ((a | 0x80000000) + 1)/2 its output and a’ is selected using mt
• s = selection of sqrt(z) and a is selected using mt
• p = (z *((z * ((z * ((z * ((z * ((z * (3.4793309169) + 7.9153501429)) - 4.0055535734)) + 2.0121252537)) - 3.2556581497)) + 1.6666667163))
• q = ((z * ((z * ((z * (7.7038154006) - 6.8828397989)) + 2.0209457874)) - 2.4033949375)) + __m128_1)
• p = p/q
• df = s & (__m128_asinf_trunc | ngt)
• q = (df
2 + z)/ (s + df)
• gt = ~(comparator a == 1)& gt
• q = (gt & q)| ( ngt & -4.37113900018624283 )
• return (((((((p * s) + q) + df) * (selection 2 and 1 using mt)) | (0x80000000 & ngt)) + ( PI & lt)) + (1.57079637050628662109375 + mt))

Figure 18: Implementation of circular arccos.

• preserve the sign and take the absolute value of the input as variable (a)
• w = comparator a > tan( PI / 8)
• x = comparator a > tan( 3 * PI / 8)
• z = ~w & x
• y = (~w & PI/2) | (z & PI/4 )
• w = (w & PI) | sign
• z= atan(a/b)
• w1 = w + z
• y = ~y2 & ((~y & pi/2) | sign)
• return selection of w and y using w2

Figure 19: Implementation of circular arctan

• first input will be known as (a) and second will be known as (b)
• w1 = comparator b < 0
• y1 = comparator a < 0
• w2 = comparator b == 0
• y2 = comparator a == 0
• sign is preserved on y1
• w1 = (w & PI) | sign
• z= atan(a/b)
• w1 = w + z
• y = ~y2 & ((~y & pi/2) | sign)
• return selection of w and y using w2

Figure 20: Implementation of circular arctan2.

• Sign is preserved and absolute value applied to input.
• ex = e ^input * input
• t = 1/ ( (0.3275911 * input) + 1)
• y = ((((((((((1.061405429 * t) + -1.453152027 ) * t ) + 1.421413741) * t ) + -0.284496736) * t ) + 0.254829592) * t )  * ex ) + 1 )
• Sign applied back to y to return approximation of erf()

Figure 21: Implementation of erf.

(Figure 20).

Distribution

The implementation of distribution functions is entirely dependent on _mm_exp_ps. In the implantation of the Gauss error function _m128 _mm_erf_ps(_m128 a) approximation uses the Handbook of Mathematical Functions (7.1.26) by Milton and Irene (1965) (Figure 21). Implementation of _m128 _mm_erfc_ps(_m128 a) subtracts a static constant of 1 by the _mm_erf_ps. For implementation of _m128 _mm_erfinv_ps(_m128 a) inlines the _mm_erf_ps, _mm_log_ps and _mm_exp_ps making the
implementation one of the lesser efficient implantation open to improvement. However, it is still faster than Intel’s implementation (Figure 22).

For implementation of _m128_mm_erfInv_ps(_m128 a) a static constant of 1 is subtracted from the input of _mm_erfinv_ps. Implementation of _m128_mm_cdfnorm_ps(_m128 a) is also known as the \( \Phi(x) \) or \( \phi(x) \). The function is the cumulative density function of a standard normal (Gaussian) random variable. It is closely related to the error function \( \text{erf}(x) \) and approximation uses the Handbook of Mathematical Functions (7.1.26) by Milton and Irene (1965) (Figure 23).

In the case of the implementation of _m128_mm_cdfnorminv_ps(_m128 a) inlining the _mm_erfinv_ps uses the equation (sqrt(2) * (2 * erfInv(2 * x) - 1)) / 2 adding 5 additional operations.

\[ x = \frac{|input|}{\sqrt{2}} \]
\[ t = \frac{1}{1 + 0.3275911 \times x} \]
\[ y = 1.0 - (((((1.061405429 \times t - 0.284496736) \times t) + 0.254829592) \times t \times \exp(x - 2)) \]
\[ \text{return } (1 + y) / 2 \text{ to approximate cdfnorm;} \]

Figure 23: Implementation of cdfnorm.

Figure 22: Implementation of erfInv.

CONCLUSION

In conclusion, the Open Source Compiler Optimized Inalienable Implementation of Intel’s SVML SIMD Intrinsic is a satisfactory alternate when software requirement calls for the use of other compilers outside of Intel’s C++. Future work will include support for other architectures and additional functionalities such as ARM and AMD.

REFERENCES

AMD64 Technology AMD64 Architecture Programmer’s Manual Volume 6:128-Bit and 256-Bit XOP and FMA4 Instructions
Intel Intrinsics Guide for SVML API
Intel® 64 and IA-32 Architectures Software Developer’s Manual
Intel® Cilk™ Plus
Newlib http://sources.redhat.com/newlib
Redhat Newlib http://sources.redhat.com/newlib