International Journal of Academic Engineering Research (IJAER)
ISSN: 2000-001X
Vol. 2 Issue 7, July – 2018, Pages: 6-13

Creating a Compiler Optimized Inlineable Implementation of Intel Svml Simd Intrinsics

Promyk Zamojski[1], Raafat Elfouly[2]
Mathematics and Computer Science Department/ Rhode Island College, RI
Email: [1]pzamojski_9958@email.ric.edu, [2]relfouly@ric.edu

Abstract: *Single Input Multiple Data (SIMD) provides data parallelism execution via implemented SIMD instructions and registers. Most mainstream computers architectures have been enhanced to provide data parallelism though 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 compared to standard SISD. Intel's SVML library offers SIMD implementation of Math 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 Math Library.

1. INTRODUCTION

Data parallelism occurs when one or more operations are applied repeatedly to several values. Generally programming languages sequential programming of scalar values. Single Input Multiple Data (SIMD) compiler intrinsics allow for operations to be grouped together which leads to an improved transistors per Floating Point Operations reducing power usage and overall modern SIMD capable CPUs. SIMD has some limitation because you cannot vectorize different functions with lots of if else statements easily. These limitation 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 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[1], a programmer is able to 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 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 write for this library to be able to be compiled by the main C/C++ compilers that support Intel Intrinsic such as GCC, CLANG and MSVC. Many Vendors have specified an API for Intrinsic such as Intel® short vector math library (SVML) [3] a software standard provided by Intel® C++ Compiler[4]. SVML functions can be broken down into 5 main category, 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 [10],[11],[12]. Reference implementation of scalar version can be translated into SIMD version from Cephes library[2]. The output of the replacement functions can be tested and compared to output form SVML library provided in Intel® C++ Compiler[4].

ORGANIZATION

This paper is organized into four section, section one is the introduction. In section two we will talk about bitwise selection. In section three we will introduce the idea of fused multiply accumulator FMA. In section four we will talk about the implementation and its parts including sign preservation, exponentiation, trigonometry, and distribution functions. Conclusion and results are in section five.

2. 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 stack-frame to a point in order to compute one value that would be common in SISD algorithms.

Figure 1: SIMD vs SISD Branching Problem

Bitwise selection occurs when vector bitmask is use to select values within vector. Vector bitmask produced when SIMD

comparative operation is is used to compare 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 allocated to each data types byte alignment.

```
__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 __svml_asinf4
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

```
radian_asin(float):
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)

It is important to understand bitwise selection in SSE SIMD operations since they will take the pace of 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 potion 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*** of the result of a ***(~mask & first_input_value)*** and the result of the ***(mask & second_input_value)***. This functions 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.

```
__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

Once a programmer understand that bitwise selection masks are used in place of conditional statements the porting of existing C/C++ library become 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 an algorithm should be evaluated as critical or noncritical to compute the final resulting value.

3. FMA INSTRUCTION SET

The FMA instruction set is an extension to the 128-bit and 256-bit Streaming SIMD Extensions 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[6]. Ultimately FMA4[9] was dropped in favor to the superior Intel FMA3 hardware implementation. FMA3 is supported in AMD processors since processors since 2014. Its goal is to remove CPU cycles by combining multiplication and addition into one instruction. This would reduce the total exaction time of an algorithm.

```
    __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

4. IMPLEMENTATION

SVML compatible library start 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 using native SSE, AVX and FMA extensions to remove unnecessary CPU cycles for SIMD implementation of math functions.

Implementation of Rounding intrinsics are able to be mapped to existing SSE intrinsics and Integer Division are able to be unpacked into scalar operations and repacked into vectors and do not need implantation.

4.1 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.

```
__m128_sign_mask  = (0x80000000)

__m128 _mm_preservesignbit_ps(const __m128 a) {
  return _mm_and_ps(a, __m128_sign_mask);
}
__m128 _mm_effectsignbit_ps(const __m128 a, const __m128 b) {
  return _mm_xor_ps(a, b);
}
```

Figure 8: Sign Preservation Functions

This will make any porting of standard C/C++ code to using Intel SIMD intrinsic much easier and more efficient at run time.

4.2 EXPONENTIATION

Starting with the *__m128 _mm_log_ps(__m128 a)* function. This is done though shifting 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: 32-bit IEEE-745 floating point number

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.

$$ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5} \ldots$$
$$ln(1+x) = \sum_{n=1}^{\infty} (-1)^{(n\mp 1)} \frac{x^n}{n} \quad (1)$$

For 64-bit versions of the Inverse Mantissa Mask remains the same however the Mantissa however the *__m128i_i32_0x7f* representing *127* or 2^{8-1} representing

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

exponent bit turns to *__m128i_i64_0x400* representing *1024* or 2^{11-1} for the 64-bit representation of *IEEE-745* floating point representation. In the case of *_mm_cvtepi32_ps* changed to *_mm_cvtepi64_pd* for 64-bit version exist inside the AVX-512[5] standard though missing inside SSE family of functions. These missing functions are able to be back ported using SSE.

```
__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),
        _mm_castpd_si128(__m128d_Int64ToDoubleMagic));
}
```

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

__m128d_Int64ToDoubleMagic numerical contrast represented by (***0x4337FFFFE5B60600)*** or (***6.755399e+15)*** special value 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 is from -2^{63} to 2^{63}. This range reduction does not have any significant impact however 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)* call the ineligible intrinsic of *_mm_log_ps multiplied* by numerical constants. In the case of *_mm_log2_ps* ***log2(exp(1))*** can be pre-computed into constant ***(1.44269504088896341F)*** then multiplied by log(x) to produce correct output. In the case of *_mm_log10_ps* ***log10(exp(1))*** into constant ***(0.4342944819032518F)*** then multiplied by *_mm_log_ps* to produce 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 builtin SSE *_mm_floor_ps* functions of the resulting value. The absolute value function just strips the sign bit example of absolute value implementation missing form SSE family.

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

Figure 12: SSE Absolute Value

For the implementation of *__m128 _mm_exp_ps(__m128 a)* takes a similar approach to the logarithm functions. Implementation modified from Cephes where approximation is calculated from constants and 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*[6] which calculates ***2^x-1*** **so an approximation is necessary for implementation.**

1. *In input range is limited using _mm_min_ps (88.3762626647949F) as the high limit and _mm_max_ps (-88.3762626647949F)*
2. *ex = x • log2(e) + ½*
where log2(e) is pre-computed as (1.44269504088896341F)
3. *ex = floor(ex)*
4. *$y = x - K_n(xf)$, with $K_n(xf) = a*x_f^n + b*x_f^{n-1} + c*x_f^{n-2} + ...$*
5. *$y = y * x^2 + x + 1$*
6. *value as the approximated exponential $e^x = y * (2^{ex})$ (IEEE745 Exponent filed and the fraction field of variable ex)*

Figure 13: Implementation of exp

In the 64-bit implementation it is the same as 32-bit implementation however the output of previous implementation of *exp* is multiplied ½ to account for doubling of precision**.** For the implementation of *__m128 _mm_exp10_ps(__m128 a)* the we can use the mathematical rule of ***exp10(x)=exp(x*log(10))*** where ***log(10)*** can be pre-computed into constant ***(2.302585092994045684 02F)*** then multiplied input value and the result of that put into exp*_mm_exp_ps(__m128 a)* to produce 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 will not work in practice. In the case of the implementation of *__m128 _mm_pow_ps(__m128 a,__m128 b)*

$$x^y = x > 0, \exp(\ln(|x|) * y) \quad (2)$$

The input of base is passed into *_mm_abs_ps* whose result is passed into *mm log ps* which is then multiple by floating point representation exponent with the result is 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 implement as followed.

$$\sqrt[root]{base} = \begin{cases} base > 0, exp(ln\frac{(|base|)}{root}) \\ base < 0, -exp(ln\frac{(|base|)}{root}) \end{cases} \quad (3)$$

The sign is persevered with *mm preservesignbit ps* and the final output we 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 we back port *_mm_rcp14_ps* intrinsic introduced in AVX-512. The AVX-512 version uses 14-bit of precision or maximum relative error for this approximation is less than ***2^{-14}***. The back ported version uses the full range. An inverse version of the nthroot function can then call the reciprocal intrinsic which then can be used to create a *_mm_cbrt_ps* and *_mm_invcbrt_ps* for 32-bit versions. In addition to *_mm_cbrt_pd* and *_mm_invcbrt_pd* for 64-bit versions. This would be done by passing a constant of 3 into the exponent parameter.

In the cases of *_mm_svml_sqrt_ps* and *_mm_invsqrt_ps* they are just alternate names for *_mm_sqrt_ps* and *_mm_rsqrt_ps.* Finally *_mm_expm1_ps* just calls *_mm_exp_ps* then subtract 1 from its result and *_mm_log1p_p*s adds 1 to the input parameter. With Exponentiation library finished Trigonometry and Distribution portions of SVML are able to be implemented.

4.3 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)*

$$\sinh(x) = \frac{e^x - e^{-x}}{2} \quad (4)$$

Value of ***exp(x)*** can be processed held in a variable then the value of ***exp(-x)*** can be translated to reciprocal of ***exp(x)*** using a builtin 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 then divided by 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)*

$$\cosh(x) = \frac{e^x + e^{-x}}{2} \quad (5)$$

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)*

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (6)$$

Value of ***exp(2* x)*** can be processed held in a variable then the value is subtracted by constant of 1 for numerator. The denominator is computed by variable added to a contact of 1 by. 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)*

$$\operatorname{arsinh}(x) = \ln(x + \sqrt{x^2 + 1}) \quad (7)$$

the value of $\mathbf{x^2+1}$ will be computed using FMA *_mm_fmadd_ps(x, x, __m128_1)* then the result of that will

be passed into built in *_mm_sqrt_ps* then added to input and finally passed to our exponentiation function *_mm_log_ps*. This implantation only using 3 additional operation to the inline *_mm_log_ps* function. In the implementation of *__m128 _mm_acosh_ps(__m128 a)*

$$\operatorname{arcosh}(x) = \ln(x + \sqrt{x^2 - 1}) \quad (8)$$

is the same as *_mm_asinh_ps however* the value of $\mathbf{x^2+1}$ is replaced by $\mathbf{x^{2--}1}$ 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)*

$$\operatorname{artanh}(x) = \frac{1}{2}\ln(\frac{1+x}{1-x}) \quad (9)$$

(1 + x / 1 - x) passed to our exponentiation function *_mm_log_ps* and multiplied by static constant of **½**. This implantation only using 4 additional operation to the inline *_mm_log_ps* function. For the implementation of *__m128 _mm_hypot_ps(__m128 a, __m128 b)*

$$hypot(a, b) = \sqrt{a^2 + b^2} \quad (10)$$

by using combination of built in SSE ***sqrt()*** in combination with FMA with SSE multiply *_mm_fmadd_ps(a, a,_mm_mul_ps(b, b))* giving us the most performance outcome. For the implementation of *__m128 _mm_sin_ps(__m128 a)* is a direct port of Cephes[2].

1. *preserve the sign and take the absolute value of the input as variable (a)*
2. *input ins multiplied by constant of 4 / PI and converted to integer stored as variable i0*
3. *i0 = (i0 + 1) + ~1*
4. *y = conversion of i0 to float*
5. *i1 = i0+4 and its result is shifted left 29 places shifting in zeros*
6. *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 signswap)*
7. *the new sign come as a result of xor of signswap by the polymask*
8. *a = ((y * -3.77489497744594108) + ((y * -2.418756849853515625) + ((y * -0.78515625)+ a)))*
9. *y = (a^2/2 + ((((((2.443315711809948 * a^2) - 1.388731625493765) * a^2) + 4.166664568298827) * a^2) * a^2)) + 1*
10. *y2 = ((((((-1.9515295891 * a^2) + 8.3321608736) * a^2) - 1.6666654611E) * a^2) * a) + a*
11. *select from y and y2 using polymask and return sign to result*

Figure 14: Implementation of circular sin

For the implementation of *__m128 _mm_cos_ps(__m128 a)* is a direct port of Cephes[2].

1. *preserve the sign and take the absolute value of the input as variable (a)*
2. *input ins multiplied by constant of 4 / PI and converted to integer stored as variable i0*
3. *i0 = (i0 + 1) + ~1*
4. *y = conversion of i0 to float*
5. *i1 = i0+4 and its result is shifted left 29 places shifting in zeros*
6. *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 signswap)*
7. *a = ((y * -3.77489497744594108) + ((y * -2.418756849853515625) + ((y * -0.78515625)+ a)))*
8. *y = (-a^2/2 + ((((((2.443315711809948 * a^2) - 1.388731625493765) * a^2) + 4.166664568298827) * a^2) * a^2)) + 1*
9. *y2 = ((((((-1.9515295891 * a^2) + 8.3321608736) * a^2) - 1.6666654611) * a^2) * a) + a*
10. *select from y and y2 using polymask and return sign to result*

Figure 15: Implementation of circular cos

The *__m128 _mm_sincos_ps(__m128 *mem_addr, __m128 a)* implementation combines **sin()** and **cos()** functions into a single function combining redundant operation are computed using the same numerical constants. For the implementation of *__m128 _mm_tan_ps(__m128 a)* direct port of Cephes[2].

1. *preserve the sign and take the absolute value of the input as variable (a)*
2. *input ins multiplied by constant of 4 / PI and converted to integer stored as variable i0*
3. *y = is the conversion back from i0 (effectively truncating)*
4. *i1 = i0 + 1 and compared == 1 where i1 becomes comparative mask*
5. *i0 = i0 + i1&1*
6. *y = y + ((i0 convert to float) & 1)*
7. *z = (y * -3.77489497744594108 + (y * -2.418756849853515625 + ((y * -0.78515625) + original input)))*
8. *y = ((((((((((9.38540185543 * z^2) + 3.11992232697) * z^2) + 2.44301354525) * z^2) + 5.34112807005) * z^2) + 1.33387994085) * z^2) + 3.33331568548) * z^2) * z) + z*
9. *z = -1/y*
10. *i1 = i0 + 2 and its result is compared == 2 making i1 a selection mask*
11. *y and z are selected based on i1 and sign is return to resulting output*

Figure 16: Implementation of circular tan

For implementations of *__m128 _mm_sind_ps(__m128 a), __m128 _mm_cosd_ps(__m128 a),* and *__m128 _mm_tand_ps(__m128 a)* multiply the input of *_mm_sin_ps, _mm_cos_ps* and *_mm_tan_ps* multiplied by static constant of **π /180.** In the implementation of *__m128 _mm_asin_ps(__m128 a)* is also a port of Cephes[2].

1. *preserve the sign and take the absolute value of the input as variable (a)*
2. *a is compared > 1/2 producing comparative mask*
3. *z = (1-a)/2*
4. *w = sqrt(z)*
5. *a = selection of a and w using comparative mask*
6. *z = selection of a^2 and z comparative mask*
7. *y = ((((((((((z * 4.2163199048) + 2.4181311049) * z) + 4.5470025998) * z) + 7.4953002686) * z) + 1.6666752422) * z) * a) + a)*
8. *z = pi/2 - (y + y)*
9. *result is selection between y and z using comparative mask and sign is return to result*

Figure 17: Implementation of circular arcsin

The implementation of *__m128 _mm_acos_ps(__m128 a)* does not call *_mm_acos_ps* since that would have to be inlined 2 times creating a performance hit. Implantation used is based around the SunPro/Sun Microsystem version modified by Ian Lance Taylor of Cygnus of found adopted into many versions of libc such as NewLib[7].

1. *input is variable of a*
2. *gt = comparison of > a and 1/2*
3. *lt = comparison of < a and -1*
4. *nlt = xor(a, 0xFFFFFFFF)*
5. *mt = ~lt & ngt*
6. *z = ((a | 0x80000000) + 1)/2 its output and a^2 is selected using mt*
7. *s = selection of sqrt(z) and a is selected using mt*
8. *p = (z *((z * ((z * ((z * ((z * ((z * 3.4793309169) + 7.9153501429)) - 4.0055535734)) + 2.0121252537)) - 3.2556581497)) + 1.6666667163))*
9. *q = ((z * ((z * ((z * ((z * 7.7038154006) - 6.8828397989)) + 2.0209457874)) - 2.4033949375)) + __m128_1)*
10. *p = p/q*
11. *df = s & (__m128_asinf_trunc | ngt)*
12. *q = (df^2 + z)/ (s + df)*
13. *gt = ~(comparator a == 1)& gt*
14. *q = (gt & q)| (ngt & -4.37113900018624283)*
15. *return ((((((p * s) + q) + df) * (selection 2 and 1 using mt)) | (0x80000000 & ngt)) + (PI & lt)) + (1.570796370506286621093 75 + mt))*

Figure 18: Implementation of circular arccos

In the implementation of *__m128 _mm_atan_ps(__m128 a)* ports from Cephes Inverse circular tangent.

```
1.  preserve the sign and take the absolute value of the input as variable (a)
2.  w = comparator a > tan( PI / 8)
3.  x = comparator a > tan( 3 * PI / 8)
4.  z = ~w & x
5.  y = (~w & PI/2) | (z & PI/4 )
6.  w = (w & -1/a) | (z & (a - 1) * 1/(a + 1))
7.  a = ((~x & a) | w)
8.  return (y + ((((((((8.05374449538 * a^2) - 1.38776856032) * a^2) + 1.99777106478) * a^2) - 3.33329491539) * a^2) * a) + a)) and return sign to output
```

Figure 19: Implementation of circular arctan

Our *__m128 _mm_atan2_ps(__m128 a, __m128 b)* implementation calling *_mm_atan_ps (a/b)* checking for all relevant corner cases. Implementing *_mm_atan_ps* before *_mm_atan2_ps instead of* using *_mm_atan2_ps(a,__m128_1)* 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.*

```
1.  first input will be known as (a) and second will be known as (b)
2.  w1 = comparator b < 0
3.  y1 = comparator a < 0
4.  w2 = comparator b == 0
5.  y2 = comparator a == 0
6.  sign is preserved on y1
7.  w1 = (w & PI) | sign
8.  z= atan(a/b)
9.  w1 = w + z
10. y = ~y2 & ((~y & pi/2) | sign)
11. return selection of w and y using w2
```

Figure 20: Implementation of circular arctan2

4.4 DISTRIBUTION

For 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)*

$$erf(x) = \frac{1}{\sqrt{(\pi)}} \int_{-x}^{x} \mathrm{e}^{-t^2} dt$$

$$erf(x) = \frac{2}{\sqrt{(\pi)}} \int_{0}^{x} \mathrm{e}^{-t^2} dt \qquad (11)$$

approximation uses the Milton Abramowitz & Irene A. Stegun Handbook of Mathematical Functions (7.1.26) [8].

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

Figure 21: Implementation of erf

Implementation of *__m128 _mm_erfc_ps(__m128 a)* subtracts 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.

1. *sign of input is preserved and absolute value of input taken*
2. $r1 = ((((((-0.140543331 * input^2) + 0.914624893) * input^2) - 1.645349621) * input^2) + 0.886226899) * input) / (((((((0.012229801 * input^2) - 0.329097515) * input^2) + 1.442710462) * input^2) - 2.118377725)* input^2) + 1)$
3. $y1 = (-ln(1 – input)) / 2$
4. $r2 = ((((((1.641345311, y1) + 3.429567803) * y1) - 1.62490649) * y1) – 1.970840454) / ((((-1.970840454 * y1) + 3.543889200)* y1) + 1)$
5. $rx = max(r1, r2)$
6. *restore the sign back to rx as well as input*
7. $r3 = (rx - ((erf(rx) - a)/ (2/(sqrt(\pi) * exp(rx^{-2})))$))
8. $return ((((erf(r3) - input) / 2) / ((r3^{-2}) * sqrt(\pi))) - r3)$

Figure 22: Implementation of erfinv

For implementation of *__m128 _mm_erficnv_ps(__m128 a)* static constant of 1 is subtracted from the input of *_mm_erfinv_ps.* For implementation of *__m128 _mm_cdfnorm_ps(__m128 a)* also known as the **Φ(x) or phi()**. The function is the cumulative density function of a standard normal (Gaussian) random variable. It is closely related to the error function **erf(x)** and approximation uses the Milton Abramowitz & Irene A. Stegun Handbook of Mathematical Functions (7.1.26) [8].

1. *Save the sign of input*
2. $x = |input| / sqrt(2)$
3. $t = 1/(1 + 0.3275911 * x)$
4. $y = 1.0 - (((((1.061405429 * t - 1.453152027)*t) + 1.421413741) * t - 0.284496736)*t + 0.254829592) * t * exp(-x^2)$
5. *restore the sign of input to y*
6. *return (1 + y)/2 to approximate cdfnorm;*

Figure 23: Implementation of cdfnorm

int the case of the implementation of *__m128 _mm_cdfnorminv_ps (__m128 a)* inlineing the *_mm_erfinv_ps* using the equation **(sqrt(2) * (2 * erfinv(2 * x) - 1)) / 2** adding 5 additional operations.

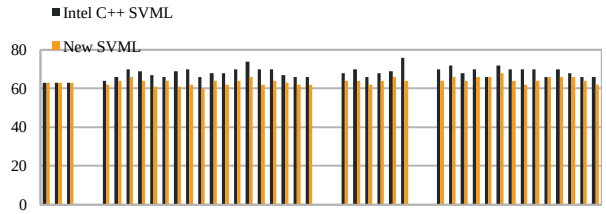
APPENDIX

For Implementation details refer to
Promyk Zamojski, SuperSIMD library source code.

https://bitbucket.org/pzam/supersimd/src
SuperSIMD/include/constants/ SuperSIMD/include/svml/

5 RESULTS

X – Axis is CPU cycles, Y – Axis are functions compared. (Lower Is Better)



From Left to Right Rounding, Trigonometric, Distribution and Exponentiation.

NOTE: CPU cycles may differ from machine to machine however should show the same relative performance gains distributed equally.

Compiler Optimized Inalienable Implementation of Intel's SVML SIMD Intrinsic are able to gain a ***(3% - 12%)*** in performance and efficiency though the reduction of CPU Cycle overhead is majority of cases. The new implementation has a less than ***0.0276%*** numerical inaccuracy due to rounding error when compared to Intel's' implantation of SVML. A 4x*(newer)*-10x*(older)* performance gain to libC SISD implementation.

CONCLUSION

In conclusion the Open Source Compiler Optimized Inalienable Implementation of Intel's SVML SIMD Intrinsic are 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 functionality such as ARM and AMD.

REFERENCES

[1] Intel® Cilk™ Plus from https://www.cilkplus.org/
[2] S. L. Moshier, Cephes library from http://www.netlib.org/cephes
[3] Intel Intrinsics Guide for SVML API https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=SVML
[4] Overview: Intrinsics for Short Vector Math Library (SVML) Functions https://software.intel.com/en-us/node/524289
[5] Overview: Intrinsics for AVX-512 Functions https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX_512
[6] Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4, December 2017 https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf
[7] Redhat Newlib http://sources.redhat.com/newlib
[8] Milton Abramowitz, Irene A. Stegun, Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables June 1, 1965.
[9] AMD64 Technology AMD64 Architecture Programmer's Manual Volume 6:128-Bit and 256-Bit XOP

and FMA4 Instructions
https://support.amd.com/TechDocs/43479.pdf
[10] Malossi, A. Cristiano I. Ineichen, Yves. Bekas, Costas. Curioni, Alessandro. 2015/01/19 Fast Exponential Computation on SIMD Architectures
[11] Nyland, Lars 2004/01/17 Fast Trigonometric Functions Using Intel's Sse2
[12] Kretz, Matthias 2015/10/16 Extending C++ for Explicit Data-Parallel Programming via SIMD Vector Types