Enhanced Vector Math Support on the Intel®AVX-512 Architecture

Cristina S. Anderson
Intel Corporation
2111 NE 25th Ave
Hillsboro, OR 97124
cristina.s.anderson@intel.com

Jingwei Zhang
Intel Corporation
2111 NE 25th Ave
Hillsboro, OR 97124
jingwei.zhang@intel.com

Marius Cornea
Intel Corporation
2111 NE 25th Ave
Hillsboro, OR 97124
marius.cornea@intel.com

***Abstract*—The Intel®AVX-512 architecture adds new capabilities such as masked execution, floating-point exception suppression and static rounding modes, as well as a small set of new instructions for mathematical library support. These new features allow for better compliance with floating-point or language standards (e.g. no spurious floating-point exceptions, and faster or more accurate code for directed rounding modes), as well as simpler, smaller footprint implementations that eliminate branches and special case paths. Performance is also improved, in particular for vector mathematical functions (which benefit from easier processing in the main path, and fast access to small lookup tables). In this paper, we describe the relevant new features and their possible applications to floating-point computation. The code examples include a few compact implementation sequences for some common vector mathematical functions.**

***Index Terms*—SIMD, vector mathematical function, floating-point**

I. INTRODUCTION

SIMD (single instruction, multiple data) architecture is an effective way to exploit data level parallelism on CPUs. The Intel® AVX-512 instruction set is the latest SIMD extension for the x86 instruction set architecture [1]. In addition to the register widening from 256-bit to 512-bit, there are architectural enhancements added in AVX-512 such as new types of SIMD instructions, conditional execution, and instruction embedded floating-point control. Here is the subset of new AVX-512 instructions discussed in this paper:

- Setting of special floating-point outputs (and exception signaling if specified)
- Cross-lane permutation with two sources
- Conversion of exponent of floating-point values to floating-point values and extraction of normalized mantissa from floating-point values
- Scaling by powers of 2, with support for special cases
- Enhanced reciprocal and reciprocal square root approximation
- Testing types of floating-point values
- Rounding to specific number of fractional bits and subsequential argument reduction
- Range restriction calculation

As a result, many vector mathematical functions can have compact, branch-free AVX-512 implementations, conforming to the IEEE Standard 754-2008 for Floating-Point Arithmetic [2]. The following sections will describe each new feature in detail, and we will show short code sequences to illustrate usage in the implementation of common vector mathematical functions.

II. NEW FEATURES OF INTEL® AVX-512 INSTRUCTION SET

A. Static Rounding Modes and Suppress-all-exception Bit

The rounding mode for a given floating-point operation on the x86 architecture is usually specified dynamically in the floating-point environment control register, except for a small number of Intel® AVX-512 round and convert instructions, e.g. *VROUNDPD*. In these cases, the rounding mode can be specified statically using an instruction encoding attribute in the instruction opcode, which can override the default rounding mode stored in the control register. The static rounding mode override in AVX-512 also implies *suppress-all-exceptions* (SAE), which treats all floating-point exceptions as masked, and in addition ensures that no floating-point status flag is changed. In the absence of SAE, there would have to be a trade-off between standard conformance [2], [3], and performance (additional steps or branches would be needed to ensure strict standard conformance). The static rounding mode feature is also used, e.g. round-toward-zero to avoid generating infinities in intermediate steps (an example will follow in Section E); a static round-to-nearest could also be useful in accuracy-sensitive applications.

B. Handling of Special Cases

Intel® AVX-512 provides *mask* registers that can be used for conditional execution. Each AVX-512 instruction in EVEX encoding can take an extra operand called *mask* register, which contains the predicate bits to control the output of each vector element. The SIMD operation on a given element will be carried out only if its corresponding *mask* predicate bit is set, otherwise, the destination element is either preserved (merge-masking) or zeroed out (zeroing-masking.) Masked execution is an efficient way of handling special cases in the main path. As an example, consider

VADDPS dest {mask}{z}, src1, [src2]{1to16}, {rz-sae}.

This instruction will first load a 32-bit element from memory using the address in *src2*, and will replicate that element 16

times to form a vector 512-bit wide, also known as *broadcasting* in AVX-512. Then, it will add that vector with*src1*, element by element, and place the result into *dest* if the corresponding *mask* bit is 1. Otherwise, it will write the value zero to *dest* as indicated by $\{z\}$ for zeroing-masking (the default is merge-masking.) The addition is performed in round-toward-zero mode, with SAE as indicated by $\{$*rz-sae*$\}$.

There is also a new "fixup" instruction that can be used to set special outputs, and optionally raise Invalid or Divide-by-Zero exceptions (the status flags are set when exceptions are masked). However, in most cases masked execution is sufficient for handling special cases. Its format is:

VFIXUPIMMPS/PD dest $\{$*mask*$\}\{z\}$*, src1, src2, imm8.*

The destination register *dest* contains the results of a function computation; these results must be correct for valid inputs in the function domain, but not necessarily for special inputs (e.g. the computation may have incorrectly produced a NaN output for an infinity input). *VFIXUPIMM* can be used to correct the outputs for special cases. The first source *src1* is treated as a 32-bit table (8 entries of 4 bits each), which defines the desired outputs for each of the following input classes: qNaN, sNaN, Zero, +1.0, $\pm\infty$, finite positive (and not 1.0), finite negative. A 4-bit table entry is read for each element, based on the class of the input *src2*. This table entry selects one of 16 possible outputs, including: unmodified destination (used when *src2* is in the function domain and *dest* already contains the desired result), destination equal to the unmodified input (e.g. to correctly set $\mathrm{sqrt}(\pm 0) = \pm 0$, qNaN Indefinite, $\pm\infty$, ± 0, and other values that can be useful for fixing up common functions, e.g. $\pi/2$ for $\mathrm{atan}(+\infty)$. The 8 bits in the immediate field *imm8* can be set to raise either Invalid, or Divide-by-Zero when *src2* is one of the following: $\pm\infty$, finite negative, sNaN, +1.0, ± 0.

C. Permute Instructions

The syntax for these instructions is:

- *VPERMI2PS/PD dest* $\{$*mask*$\}\{z\}$*, src1, src2.*
  Permute elements in *src1* and *src2*, using indices from *dest*; *dest* is overwritten by the results
- *VPERMT2PS/PD dest* $\{$*mask*$\}\{z\}$*, src1, src2.*
  Permute elements in *src2* and *dest*, using indices from *src1*; *dest* is overwritten by the results
- *VPERMPS/PD dest* $\{$*mask*$\}\{z\}$*, src1, src2.*
  Permute elements in *src2* using indices in *src1*; store results in *dest*

Lookup tables are frequently used in mathematical library design [4]; however, poor vector gather support limits their benefit for vector functions. A small table lookup can be implemented using one of the *VPERM* instructions. The size of the table would be 32 single or 16 double precision elements for *VPERMI2* and *VPERMT2*, and 16 single or 8 double for *VPERM*. Larger table lookup can be implemented with a sequence of permute and blend instructions. These instructions can be a lot less expensive than hardware vector gather operation in term of performance.

D. Exponent and Mantissa Extraction

The new *VGETEXP* instruction returns the unbiased exponent of the input, in floating-point format. It is implemented as $VGETEXP(x) = \mathrm{floor}(\log 2(|x|))$ for all inputs, including subnormals and special cases. The associated instruction *VGETMANT* normalizes the mantissa to one of the four following intervals, according to bits in the immediate field: [1, 2), [0.5, 1), [0.75, 1.5), and [0.5, 2), where the last one takes into account whether the exponent is even or odd. Other bits in the immediate field can be used to allow *VGETMANT* to raise Invalid exception on negative inputs, and to specify whether the sign of the input should be preserved in the output.

VGETEXPPS/PD dest $\{$*mask*$\}\{z\}$*, src1.*

VGETMANTPS/PD dest $\{$*mask*$\}\{z\}$*, src1, imm8.*

The following identity holds for these two instructions: $x = 2^{VGETEXP(x)} * VGETMANT(x, 0)$, where 0 in the immediate field means the mantissa is normalized to [1, 2), the sign of x is preserved, and no exception is raised.

In Example 1, $\log 2(x)$ approximation accurate to 21 bits is computed using piecewise polynomial interpolation. The polynomial coefficients for each of the 16 intervals considered are stored in separate tables, and accessed with *VPERMPS* instructions. The mantissa is normalized to $m_x \in [0.75, 1.5)$, and the leading mantissa bits serve as table indices for the coefficient tables. The polynomial approximates $\log 2(m_x) = \log 2((m_x - 1) + 1)$.

The *VGETMANT* immediate field also requests that the Invalid exception be signaled when the input is negative; the *VGETMANT* result is set to qNaN Indefinite in that case. *VGETMANT* returns 1.0 for $\pm\infty$ and ± 0, while *VGETEXP* returns $+\infty$ and $-\infty$ for $\pm\infty$ and ± 0 respectively. Both instructions generate NaNs for NaN inputs. This ensures that special cases are treated correctly, without additional steps. The only standard requirement that is not met in Example 1 is signaling Divide-by-Zero for $\log 2(\pm 0)$, which could be solved by adding a *VFIXUPIMM* operation at the end.

```
; input, output in zmm0
vgetmantps zmm1, zmm0, 0bh ;  mantissa mx in [3/4, 3/2)
vgetexpps zmm2, zmm0       ;  exponent
vpsrld zmm3, zmm1, 23-4    ;  index for coefficient tables
vsubps zmm1, zmm1, [One]{1to16} ; reduction r=mx-1
vpermps zmm14, zmm3, ZMMWORD PTR [C4] ; c4
vpermps zmm13, zmm3, ZMMWORD PTR [C3] ; c3
vpermps zmm12, zmm3, ZMMWORD PTR [C2] ; c2
vpermps zmm11, zmm3, ZMMWORD PTR [C1] ; c1
vmulps zmm5, zmm1, zmm1               ; r^2
vfmadd213ps    zmm14, zmm1, zmm13     ; c3+c4*r
vpermps zmm4, zmm3, ZMMWORD PTR [Exp] ; exponent adjustment
vfmadd213ps    zmm12, zmm1, zmm11     ; c1+c2*r
vaddps zmm0, zmm2, zmm4               ; adjusted exponent k
; p = (c1+c2*r)+r^2*(c3+c4*r)
vfmadd213ps    zmm14, zmm5, zmm12
vfmadd231ps    zmm0, zmm1, zmm14      ; result = k + r*p
```

Example 1. SIMD single precision $\log 2(x)$ accurate to 21 bits

E. Scaling by Powers of 2

The new instruction is

VSCALEFPD/PS dest $\{$*mask*$\}\{z\}$*, src1, src2,*

which computes *dest* $= src1 * 2^{\lfloor src2 \rfloor}$. Overflow and Underflow exceptions are treated as described in the IEEE Standard 754-2008 [2], even though this is not an IEEE-defined operation.

Special case behavior is carefully defined, in a manner that ensures that *VSCALEF* can function as a last step as well as a fixup instruction in implementations of functions such as $\exp()$ or $\mathrm{pow}()$ (meaning that no other steps are needed to set the correct outputs for special inputs, e.g. $\pm\infty$). A branch-free double precision $\exp()$ computation accurate to 50 bits is shown in Example 2. $\exp(x)$ is formed as $\exp(x - Z_0\log(2))*\exp(Z_0\log(2)) = e^R*2^{Z_0} = e^R*2^{Z_0-\lfloor Z_0\rfloor}*2^{\lfloor Z_0\rfloor}$, where $R = x - Z_0\log(2)$ and Z_0 is $x/\log(2)$ rounded down to 4 fractional bits. e^R is calculated through polynomial approximation, $2^{Z_0-\lfloor Z_0\rfloor}$ through table-lookup, and a final *VSCALEF* instruction is used to scale their product by $2^{\lfloor Z0\rfloor}$ and to generate the final result. *VSCALEF* also yields correct results for special $\exp()$ cases ($\pm\infty$ and NaNs). Infinity inputs to our computation sequence generate an intermediate NaN result, which is then passed as the first argument to *VSCALEF*. However, *VSCALEF* is defined so that *VSCALEF*(qNaN, $+\infty$) = $+\infty$, and *VSCALEF*(qNaN, $-\infty$) = 0. Inputs that are very large in magnitude are also handled successfully in the main path: spurious overflow to infinity (which would later lead to an intermediate NaN result) is avoided by executing $x/\log(2)$ in round-toward-zero rounding mode, and other unwanted Overflow/Underflow exceptions in the polynomial computation are avoided by forcing the reduced argument to a limited range. By ensuring a finite and positive result for the $\exp(R)*2^{\mathrm{frac}(Z_0)}$ computation, the final *VSCALEF* instruction can correctly generate overflow/underflow results for all out-of-range inputs. Spurious status flags (such as Invalid for infinity inputs) are avoided by using SAE for the affected operations.

```
; input, output in zmm0
; 2^(52-4)*1.5 + x*log2(e), RZ mode
vmovapd zmm3, ZMMWORD PTR [Log2E]
vmovapd zmm4, ZMMWORD PTR [Shifter]
vfmadd213pd zmm3, zmm0, zmm4, {rz-sae}
; Z0 ~= x*log2(e), rounded down to 4 fractional bits
vsubpd zmm5, zmm3, zmm4
; R = x - Z0*log(2), SAE on
vfnmadd231pd zmm0, zmm5, [Ln2Hi]{1to16}, {rn-sae}
vfnmadd231pd zmm0, zmm5, [Ln2Lo]{1to16}, {rn-sae}
; Table lookup: T ~= 2.0^f, where f = Z0 - floor(Z0)
vmovapd zmm10, ZMMWORD PTR [ExpTbl]
vpermt2pd zmm10, zmm3, ZMMWORD PTR [ExpTbl+64]
; mask exponent bit to ensure |R|<2 even for special cases
vandpd zmm0, zmm0, [EMask]{1to16}
<Polynomial computation (of degree 6) omitted where
P (zmm11) ~= exp(R)-1>
vfmadd213pd zmm11, zmm10, zmm10 ; T+T*P
vscalefpd zmm0, zmm11, zmm5     ; 2^floor(Z0)*(T+T*P)
```

Example 2. SIMD double precision $\exp(x)$ accurate to 50 bits

Some other examples of *VSCALEF*, *VGETMANT*, *VGETEXP* usage are:

- Software division, where $a/b = m_a2^{e_a}/m_b2^{e_b} = (m_a/m_b)*2^{e_a-e_b}$ with m_x = *VGETMANT*$(x,0)$, e_x = *VGETEXP*(x), and the final scaling through *VSCALEF*. This reduction allows for a branch-free implementation of division, which covers overflow, underflow, and also special inputs (zeroes, infinities, subnormals). $|m_x|$ is in [1,2) for all non-NaN inputs. m_a/m_b can be computed to the desired accuracy with Newton-Raphson iterations. The SAE feature can help ensure spurious flag settings do not occur. Flags can be set correctly as part of the computation (except for Divide-by-Zero, which may require an additional step).
- x^α, where α is constant (e.g. $\alpha = 1/3$ for the cube root function). The basic reduction for this computation is: $x^\alpha = (m_x2^{e_x})^\alpha = (m_x)^\alpha 2^{e_x\alpha} = (m_x)^\alpha 2^{e_x\alpha-\lfloor e_x\alpha\rfloor}2^{\lfloor e_x\alpha\rfloor}$.

F. Approximation Instructions

Approximations for the reciprocal and reciprocal square root functions provided by the following instructions can be refined easily to higher accuracy, and can often be used to get software implements of division, reciprocal, square root, and reciprocal square root with better throughput performance than hardware.

VRCP14PS/PD dest {*mask*}{*z*}*, src1.*

VRSQRT14PS/PD dest {*mask*}{*z*}*, src1.*

The new instructions have an accuracy of at least 14 bits (the relative error of the approximation is less than 2^{-14}), and treat subnormals correctly (unlike the legacy instructions *RCPPS* and *RSQRTPS*). They are offered in both single and double precision, where double precision computations benefit the most, since double precision reciprocal and reciprocal square root approximation instructions were not available before. The increased accuracy (from 11.5 bits to 14 bits) is sufficient to eliminate one Newton-Raphson iteration from a 50-52 bit reciprocal calculation.

For some mathematical functions, a rounded *VRCP14PD* result can be used in place of an expensive reciprocal table lookup. The same technique could be used before via the legacy *RCPPS*, however, less efficient due to the lack of *RCPPD*. Examples of functions that can benefit are logarithm functions (discussed in more detail in Section H) and the cube root functions. For cube root, $(m_x)^{1/3}$ with $m_x \in [1,2)$ can be rewritten as $(m_x)^{1/3} = (m_x * \mathit{RCP}(m_x))^{1/3} * (\mathit{RCP}(m_x))^{-1/3} = (1 + (m_x * \mathit{RCP}(m_x) - 1))^{1/3} * (\mathit{RCP}(m_x))^{-1/3} = (1+r)^{1/3} * (\mathit{RCP}(m_x))^{-1/3}$, where *RCP*$(m_x)$ is the reciprocal approximation of m_x rounded to a fixed number of fraction bits using *VRCP14* and *VRNDSCALE* instructions. $(1+r)^{1/3}$ is calculated with polynomial approximation of r and $(\mathit{RCP}(m_x))^{-1/3}$ is usually obtained by table lookup.

G. Testing of Floating-Point Categories

This is achieved with the new instruction *VFPCLASS*:

VFPCLASSPS/PD dest {*mask*}{*z*}*, src1, imm8.*

VFPCLASS checks whether the input belongs to any of the special floating-point classes specified in the immediate field *imm8*. The instruction sets a bit in the *dest* mask for each element in the input register, to indicate whether it is in any one of the special classes it was checked against. The different classes that can be checked for are sNaN, qNaN, $+0$, -0, $+\infty$, $-\infty$, subnormal, and finite negative value. The user can check for one or more of these classes, by setting the corresponding bits in the *imm8* field. *VFPCLASS* is used to detect special cases so they can be directed to a special path, or alternatively, handled with masked operations in the main path. Two examples are shown below.

```
; input, output  in zmm0
vrcp14pd zmm1, zmm0    ; R0 ~= 1/x
vmovapd zmm4, zmm0     ; copy x
vfnmadd213pd zmm0, zmm1, [One]{1to16}, {rn-sae} ; e0=1-x*R0
vfpclasspd k2, zmm1, 1eh      ; x +/-Inf or +/-0?
vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}   ; R1 = R0+R0*e0
knotw k3, k2     ; non-special mask (k3 = NOT k2)
vfnmadd213pd zmm4, zmm0, [One]{1to16}, {rn-sae} ; e1=1-x*R1
; special cases: return RCP14(x) if k2=1
vblendmpd zmm0 {k2}, zmm0, zmm1
; return result of computation if k3=1
vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-sae} ; R1+R1*e1
```

Example 3. SIMD double precision $1/x$ accurate to 52 bits

1) Reciprocal Sequence, Square Root Sequence: The reduced argument for the $1/x$ computation is $e = 1 - x * VRCP14(x)$. This expression evaluates to NaN when x is ± 0 or $\pm\infty$, as *VRCP14* returns the correct result for these special cases. *VFPCLASS* allows the programmer to set a mask value for $x = \pm 0$ or $\pm\infty$, and to clear the mask for all other x. One can then use this mask to select between the *VRCP14* output (result for special cases), or the result of a reciprocal refinement computation starting with *VRCP14* (for typical inputs). Example 3 shows a double precision reciprocal computation that follows the steps outlined above. In a similar manner, a square root computation based on *VRSQRT14* would use *VFPCLASS* to create a mask for $x = \pm 0$ or $x = +\infty$ (the result is equal to the input in these special cases).

2) Power function: The main path of $\mathrm{pow}(x, y) = 2^{y \log 2(x)}$ does not work when $x \leq 0$, $x = \infty$ or NaN, or $y = \infty$ or NaN. One *VFPCLASS* can be used to set xSpecialMask to 1 for $x \leq 0$ or $x = \infty$ or NaN. A second *VFPCLASS* would be used to set ySpecialMask to 1 for $y = \infty$ or NaN. A branch to a secondary path is taken when either mask is set.

H. Enhanced Rounding and Fraction Computation

The new instructions used for this purpose are VRNDSCALE and VREDUCE:

VRNDSCALEPS/PD dest {*mask*}{*z*}*, src1, imm8,*
VREDUCEPS/PD dest {*mask*}{*z*}*, src1, imm8.*

The *VRNDSCALE* instruction rounds the input to integer, plus M fraction bits, with result stored in floating-point format. The operation of *VRNDSCALE(src, imm8)* is equivalent to roundToInt$(src \cdot 2^M)2^{-M}$, performed as if with unbounded exponent (i.e. there is no overflow), where M is an integer retrieved form the upper 4 bits of *imm8* and the lower 4 bits of *imm8* specifies rounding and exception reporting controls similar to the legacy *VROUND* instruction. The operation of *VREDUCE(src, imm)* is equivalent to *src* $-$ *VRNDSCALE(src, imm8)*. *VRNDSCALE* can be used in argument reduction step for the $\log()$ function. The argument reduction for $\log(x)$, where $1 \leq x < 2$, can be done like this:

```
y = RCP14(x);           // y is in [0.5, 1]
y0= RNDSCALE(y, k<<4); // y0 has k mantissa bits
R = x*y0-1;             // |R|<2^(-14)+2^(-k)
```

We have now $\log(x) = -\log(y_0) + \log(1 + R)$, where $\log(1 + R)$ can be computed via polynomial approximation, and $\log(y_0)$ can be retrieved from a lookup table of $2^{k-1} + 1$ elements.

The most significant benefit of *VREDUCE* is latency reduction in common transcendental functions such as $\exp 2()$ and $\mathrm{pow}()$. Uses in other transcendental functions such as $\mathrm{atan}()$ are also possible. Example 4 shows such usage in a single precision $\exp 2()$ implementation. As with other code examples, all inputs are handled correctly in the main path (including special cases.)

```
;  input, output in zmm0, K = floor(x)
vreduceps zmm3, zmm0, 0x41 ; reduction r=x - K.b1b2b3b4
vaddps zmm1, zmm0, [C]{1to16}, {rd-sae} ; C=1.5*2^{23-4}
vbroadcastps zmm2, [C3] ; c3
vfmadd213ps zmm2, zmm3, [C2]{1to16} ; c3*r+c2
vpermps zmm4, zmm1, ZMMWORD PTR [T] ;table for 2^(.b1b2b3b4)
vmulps  zmm1, zmm4, zmm3   ; T*r
vfmadd213ps zmm2, zmm3, [C1]{1to16} ; c3*r^2+c2*r+c1
vfmadd213ps zmm2, zmm1, zmm4 ; T+T*r*(c3*r^2+c2*r+c1)
vscalefps  zmm0, zmm2, zmm0  ; scale by 2^K
```

Example 4. SIMD single-precision $\exp 2(x)$ accurate to 22 bits

I. Range Restriction Calculation

The range restriction calculation instructions *VRANGEPS/PD* in AVX-512 can be seen as an extension of the legacy *MAXPS/PD* and *MINPS/PD* instructions. The new instructions add a control field *imm8* to select a comparison operation out of minNum, maxNum, minNumMag, and maxNumMag according to the IEEE Standard 754-2008 [2], using *imm8*[1:0] and the sign of the output using *imm8*[3:2]. The instruction format is:

VRANGEPS/PD dest {*mask*}{*z*}*, src1, src2, imm8.*

An example of using such an instruction other than in the IEEE comparison operations mentioned above, could be in the Fast2Sum [5] algorithms, when the relative order of two numbers to be summed is not known in advance.

The Fast2Sum algorithm below requires $|a| \geq |b|$, which could be achieved by using *VRANGE* instruction to select $a = \mathrm{maxNumMag}(a, b)$ and $b = \mathrm{minNumMag}(a, b)$.

$$s = (a + b)_{rn},$$
$$z = (s - a)_{rn},$$
$$t = (b - z)_{rn}.$$

III. PERFORMANCE EVALUATION AND CONCLUSIONS

Most common functions in the vector mathematical library from the Intel® Math Kernel Library were optimized for AVX-512, using the new features presented in this paper, with their performance numbers published online [6].

Throughput-oriented sequences such as those used in the software-pipelined vector mathematical library typically achieve the highest speedup ratios with AVX-512 instructions. This is no surprise, since the new instructions were primarily designed to reduce the total number of operations in numerical computations; the effect on latency is less pronounced. It is also not unexpected that higher accuracy sequences (requiring more complex computations) tend to benefit more from the new instructions. Implementations using simple arithmetic

evaluations (such as polynomials) and no bit manipulations have little opportunity for further optimization.

TABLE I

Function	Single precision		Double precision	
AVX-512 new instructions	w/o	with	w/o	with
exp2	.83	.59	1.68	1.13
exp	.84	.67	1.69	1.33
log2	1.07	.81	2.55	2.31
log	.98	.82	2.46	2.30
pow	5.30	3.49	9.45	6.74
cbrt	2.61	1.34	3.47	3.07

Table I compares the performance of selected mathematical functions having ulp errors of at most 4 ulps (units-in-the-last-place), with and then without using the new AVX-512 instructions. The performance numbers are in clocks per element, for vectors of 1024 elements (the SIMD code are not unrolled), on Intel® Xeon® Platinum 8180 CPU @ 2.50GHz.

REFERENCES

[1] Intel® Architecture Software Developers Manual, https://software.intel.com/en-us/articles/intel-sdm.

[2] 754-2008 - IEEE Standard for Floating-Point Arithmetic http://standards.ieee.org/findstds/standard/754-2008.html.

[3] ISO/IEC 9899:2011 - Programming languages C https://www.iso.org/standard/57853.html.

[4] Ping Tak Peter Tang, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," ACM Transactions on Mathematical Software, 16(4):378-400, 1990.

[5] T. J. Dekker, "A floating-point technique for extending the available precision," Numerische Mathematik, 18(3):224-242, 1971.

[6] Intel® Math Kernel Library 2018 Vector Mathematics Performance and Accuracy Data https://software.intel.com/sites/products/documentation/doclib/mkl/vm/vmdata.htm.