VECTOR MATH LIBRARIES

FEATURES, USAGE AND PERFORMANCE IMPROVEMENTS

MARIUS CORNEA, CRISTINA ANDERSON, ANDREY KOLESOV, POUYA DORMIANI, ROBIN GILBERT, AHMET AKKAS

Architectural Support for Vector Math Libraries

Vector math libraries were first enabled by **Intel® SSE** (Streaming SIMD extensions: 128-bit integer and 4-way single precision instructions)

- Better throughput by exploiting data parallelism

Intel® SSE2 added support for double precision (128-bit, 2-way)

Intel® AVX (Intel® Advanced Vector Extensions): 256-bit floating point instructions, but not integer

Intel® AVX2 adds 256-bit SIMD integer operations, fused multiply-add (FMA)

Intel® AVX-512 supports 512-bit SIMD, adds several new instructions to speed up vector math functions

Intel® SIMD ISA Evolution

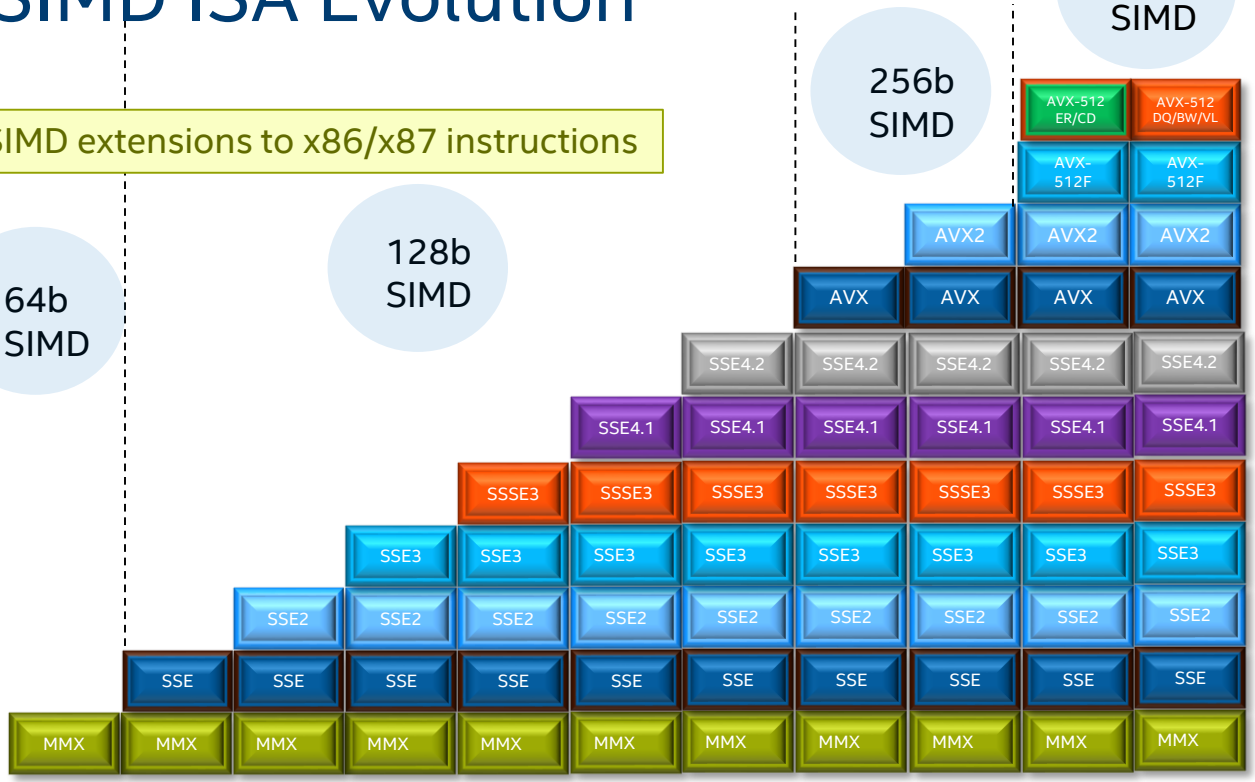
SIMD extensions to x86/x87 instructions

64b SIMD

128b SIMD

256b SIMD

512b SIMD



Performance Bottlenecks in Vector Math Libraries

Operations that cannot be supported in SIMD fashion:

- Branches
  - Some can be avoided by merging paths/blending results
    - performance impact is still significant, especially when blended path is long
  - Infrequent special or difficult cases are often redirected to a separate path (that may be scalar)
    - This allows for better performance in the main path, used by common cases
- Table Lookups
  - Technique improves performance in scalar mode; higher accuracy vector code often benefits (in spite of limitations)
    - In the absence of vector gather, lookups for each element are performed sequentially
    - Some hardware vector gather implementations are still rather slow
- **Intel® AVX** : integer operations not supported at full SIMD width

Example: Double Precision Exp2 Function (2^x, 4-ulp)

Basic Algorithm (assumes no overflow/underflow): x = round_to_int(x) + seven_frac_bits + R = dN + R

Example: if x = $1.101111101101111 \cdot 2^3$ then round_to_int(x) = 1101, seven_frac_bits = 0.1111011, R = 0.00000000111, and $2^x = 2^{round\_to\_int(x)} \cdot 2^{seven\_frac\_bits} \cdot 2^R = 2^{round\_to\_int(x)} \cdot T \cdot (poly \cdot R + 1)$, where $poly=(2^R-1)/R$

```
// round input x to integer + 7 fractional bits -> dN
tmp.f = x + Shifter;   // Shifter = 1.5 · 2^45
dN = tmp.f – Shifter;   // input rounded to nearest = round_to_int(x) + seven_frac_bits

index = tmp.i & IndexMask; // table index formed by leading 7 fractional bits, IndexMask=0x7f
tmp.i = ((~IndexMask) & tmp.i) << (52-K);  // exp. of result: 2^round_to_int(x); tmp treated as integer; K=7

T = Tbl_exp[index];  // lookup table value
R = x – dN;  // reduced argument
poly = (((c3 · R+c2) · R + c1) · R+c0); // evaluate polynomial for (2^R-1)/R, based on reduced arg. R

res.f = poly · R · T + T;   // this is T · 2^R = 2.0^(x – round_to_int(x)) = 2.0^fraction(x)
res.i = res.i + tmp.i;   // result is scaled by 2.0^round_to_int_(x), by adding correct value to exponent field
```

Exp2: Scalar versus 2-way SIMD (Intel® SSE2)

```
// x in xmm2 and xmm3
addsd    [Shifter], %xmm5 // tmp=x+Shifter->xmm5
movaps   %xmm5, %xmm2 // x+Shifter->xmm2
movq     [IndexMask], %xmm6 // IndexMask->xmm6
subsd    [Shifter], %xmm2 // dN=(x+Shifter)-Shifter->xmm2
movdqa   %xmm6, %xmm1 // IndexMask->xmm1
pandn    %xmm5, %xmm6  // tmp&~IndexMask->xmm6
pand     %xmm5, %xmm1 // index=tmp&IndexMask->xmm1
psllq    $45, %xmm6 // 2^(round_to_int(x))->xmm6
movd     %xmm1, %r8d  // extract index: index->r8d
subsd    %xmm2, %xmm3 // R=x-dN->xmm3
movsd    [c3], %xmm7 // c3->xmm7
mulsd    %xmm3, %xmm7 // c3·R->xmm7
shll     $3, %r8d // 8·index->r8d (each table entry is 8 bytes long)
addsd    [c2], %xmm7 // c2+c3·R->xmm7
mulsd    %xmm3, %xmm7 // c2·R+c3·R^2->xmm7
movq     Tbl_exp(%r8), %xmm4 // table lookup: Tbl_exp[index]->xmm4
addsd    [c1], %xmm7 // c1+c2·R+c3·R^2->xmm7
mulsd    %xmm3, %xmm7 // c1·R+c2·R^2+c3·R^3->xmm7
mulsd    %xmm4, %xmm3  // R·T->xmm3
addsd    [c0], %xmm7 // poly=c0+ c1·R+c2·R^2+c3·R^3->xmm7
mulsd    %xmm3, %xmm7 // poly·R·T->xmm7
addsd    %xmm4, %xmm7 // res.f=T+poly·R·T->xmm7
paddq    %xmm6, %xmm7  // final scaling: 2^(round_to_int(x))+res->xmm7
```

```
movaps   [Shifter], %xmm10
movaps   [c3], %xmm9
addpd    %xmm8, %xmm10 // x+Shifter
movaps   %xmm10, %xmm7
movdqa   [IndexMask], %xmm6
subpd    [Shifter], %xmm7  // dN
subpd    %xmm7, %xmm8  // R
mulpd    %xmm8, %xmm9
addpd    [c2], %xmm9
mulpd    %xmm8, %xmm9
addpd    [c1], %xmm9
andps    %xmm10, %xmm6
movdqa   [IndexMask], %xmm11
movd     %xmm6, %r8d  //extract index
pandn    %xmm10, %xmm11
psllq    $45, %xmm11   // final exponent
pextrw   $4, %xmm6, %r9d   // extract index
mulpd    %xmm8, %xmm9
shll     $3, %r8d
shll     $3, %r9d
movq     Tbl_exp(%r8), %xmm12 // lookup (low value)
movhpd   Tbl_exp(%r9), %xmm12 // lookup (high value)
mulpd    %xmm12, %xmm8
addpd    [c0], %xmm9
mulpd    %xmm8, %xmm9
addpd    %xmm9, %xmm12
paddq    %xmm11, %xmm12 // final scaling
```

Exp2: Scalar versus 4-way SIMD (Intel® AVX2)

```
vmovsd   [c3], %xmm10
vaddsd   [Shifter], %xmm2, %xmm8   // x+Shifter
vmovq    [IndexMask], %xmm7
vsubsd   [Shifter], %xmm8, %xmm3   // dN
vpand    [IndexMask], %xmm8, %xmm1   // index
vpandn   %xmm8, %xmm7, %xmm9
vmovd    %xmm1, %r8d  // extract index
vpsllq   $45, %xmm9, %xmm11  // final exponent
vsubsd   %xmm3, %xmm2, %xmm4  // R
vfmadd213sd [c2], %xmm4, %xmm10
vfmadd213sd [c1], %xmm4, %xmm10
shll     $3, %r8d
vmovq    Tbl_exp(%r8), %xmm6  // T (lookup value)
vfmadd213sd [c0], %xmm4, %xmm10  // poly
vmulsd   %xmm6, %xmm4, %xmm5   // R*T
vfmadd213sd %xmm6, %xmm5, %xmm10 //T + R*T*poly
vpaddq   %xmm11, %xmm10, %xmm12 // final scaling
```

```
vaddpd   [Shifter], %ymm2, %ymm3   // x+Shifter
vsubpd   [Shifter], %ymm3, %ymm1   // dN
vsubpd   %ymm1, %ymm2, %ymm15  // R
vandps   [IndexMask], %ymm3, %ymm9   //index
vmovapd  [c2], %ymm1
vfmadd231pd [c3], %ymm15, %ymm1
vpandn   %ymm3, %ymm7, %ymm3
vfmadd213pd [c1], %ymm15, %ymm1
vpsllq   $45, %ymm3, %ymm2   // final exponent
vfmadd213pd [c0], %ymm15, %ymm1 // polynomial
vextracti128 $1, %ymm9, %xmm12
vmovd    %xmm9, %r8d  // extract individual indices
vmovd    %xmm12, %r10d
shll     $3, %r8d
vpextrd  $2, %xmm9, %r9d
shll     $3, %r10d
vpextrd  $2, %xmm12, %r11d
shll     $3, %r9d
vmovq    Tbl_exp(%r8), %xmm10   // lookup
shll     $3, %r11d
vmovq    Tbl_exp(%r10), %xmm13  // lookup
vmovhpd  Tbl_exp(%r9), %xmm10, %xmm11   // lookup
vmovhpd  Tbl_exp(%r11), %xmm13, %xmm14  // lookup
vinsertf128 $1, %xmm14, %ymm11, %ymm9  // T
vmulpd   %ymm15, %ymm9, %ymm15 // R*T
vfmadd213pd %ymm9, %ymm15, %ymm1 // R*T*poly+T
vpaddq   %ymm2, %ymm1, %ymm10  // final scaling
```

Intel® AVX-512 Transcendental Support

- New features:
  - **Vector masks**
    - For conditional execution
    - Enable handling of some special cases in the main path
  - **Static rounding modes**, suppress-all-exceptions (**SAE**)
    - Improve standard conformance without loss of performance

- New instructions for transcendental support
  - **Permute instructions**, equivalent to small table lookups
    - Up to 32-element table (single precision), 16-element (double precision)
    - Significantly faster than vector gather
  - Extraction of normalized mantissa, exponent: **VGETMANT, VGETEXP** (e.g. used in log, pow)
  - Floating-point scaling: **VSCALEF** (e.g. used in exp, pow)
  - Better reciprocal, reciprocal square root approximations: **VRCP14** (e.g. in log, cbrt), **VRSQRT14** (e.g. in asin/acos, sqrt)
  - Special case handling: **VFPCLASS** (e.g. pow), **VFIXUPIMM**
  - Rounding and fraction calculation: **VRNDSCALE** (e.g. log, sinpi) , **VREDUCE** (e.g. exp2)
  - IEEE min/max/minabs/maxabs: **VRANGE** (e.g. high-accuracy computation – Fast2Sum)

Exp2: Scalar versus SIMD (Intel® AVX-512)

- Unlike before, new instructions allow handling of all inputs in the main path (including overflow, underflow, zero/Inf/NaN)
- Need to use a somewhat longer polynomial (due to a smaller look-up table), but the implementation is overall faster due to vector gather replacement

```
// input x in xmm0
vaddsd {rd-sae} , [Shifter], %xmm0, %xmm2
vreducesd $65, {sae}, %xmm0, %xmm5 // R
// above: R left after rounding x to 6 fractional bits
vpand    [IndexMask], %xmm2, %xmm4   // index
vmovd    %xmm4, %r8d
andl     $15, %r8d
vmovsd   Tbl_exp(,%r8,8), %xmm13 // table lookup (T)
vfmadd213sd {rn-sae}, [p5], %xmm5, %xmm6
vmulsd {rn-sae}, %xmm5, %xmm13, %xmm12 // T*R
vfmadd213sd {rn-sae}, [p4], %xmm5, %xmm6
vfmadd213sd {rn-sae}, [p3], %xmm5, %xmm6
vfmadd213sd {rn-sae}, [p2], %xmm5, %xmm6
vfmadd213sd {rn-sae}, [p1], %xmm5, %xmm6   // poly
// poly*R*T+T
vfmadd213sd {rn-sae}, %xmm13, %xmm12, %zmm6
// result scaled by 2^floor(x)
vscalefsd {rn-sae}, %zmm0, %zmm6, %zmm14
```

```
// input x in zmm15
vreducepd $65, {sae}, %zmm15, %zmm11  // R
// above: R left after rounding x to 6 fractional bits
vaddpd   {rd-sae}, [Shifter], %zmm15, %zmm10
vmovaps  [p5], %zmm14
vfmadd231pd {rn-sae}, [p6], %zmm11, %zmm14
vfmadd213pd {rn-sae}, [p4], %zmm11, %zmm14
vfmadd213pd {rn-sae}, [p3], %zmm11, %zmm14
vfmadd213pd {rn-sae}, [p2], %zmm11, %zmm14
vfmadd213pd {rn-sae}, [p1], %zmm11, %zmm14   // poly
vpandq   [IndexMask], %zmm10, %zmm13   // index
vpermi2pd %zmm0, %zmm2, %zmm13 // table lookup (T)
vmulpd {rn-sae}, %zmm11, %zmm13, %zmm12  // R*T
vfmadd213pd {rn-sae}, %zmm13, %zmm12, %zmm14
// result scaled by 2^floor(x)
vscalefpd {rn-sae}, %zmm15, %zmm14, %zmm16
```

Exp2 Measured Performance

Measured performance in cycles per element (CPE):

	Scalar	SIMD	Unrolled once	Unrolled twice
Intel® **SSE2**	9.5	5.8 (2-way)	5.5 (4-way)	**4.7** (8-way)
Intel® **AVX**	9.5	4.4 (4-way)	**4.0** (8-way)	non-optimal
Intel® **AVX2**	6.5	2.8 (4-way)	**2.7** (8-way)	**2.7** (16-way)
Intel® **AVX-512 (no new instructions)**	6.5	1.2 (8-way)	**1.1** (16-way)	non-optimal
Intel® **AVX-512 (w/ new instructions)**	8.25	0.87 (8-way)	**0.78** (16-way)	non-optimal

SVML: Intel® Compiler Short Vector Math Library

SVML contains performance-oriented transcendental math functions using a “short” vector API

- Includes the most frequently used math functions: division, reciprocal, root, exponential, logarithmic, trigonometric, error, rounding
- SIMD registers as input/output
- Custom internal ABI
- Primary purpose: Intel® Compiler auto-vectorizer support; *mostly* automatic vectorization of loops
- Additional external intrinsic-like direct call API
- Multiple accuracy levels: High, Medium (default), Low
- Feature-based internal dispatcher

SVML external API examples:

```
__m128 _mm_exp_ps(__m128 arg);
__m256 _mm256_log_ps(__m256 arg);
__m512 _mm512_invsqrt_ps(__m512 arg);
__m512d _mm512_pow_pd(__m512d argX, __m512d argY);
```

Differences Between LIBM and SVML

LIBM: Intel® Math Library, the standard math library for the compiler

- Scalar processing: one input – one output
- Optimized for shorter latency
- Maximum error < 1ulp in most cases
- Has support for all IEEE 754 rounding modes (max error < 2 ulps in directed rnd)
- Intended for correct exception-handling; sets errno

SVML: SIMD-based, for vectorized loops; contains the most commonly used functions from LIBM

- SIMD register inputs with packed 2, 4, 8 or 16 values; SIMD register output
- Written with SIMD instructions, for increased throughput
- Maximum error: 1ulp, 4ulp and half-mantissa (supports multiple levels)
- Works only in RNE (rounding to nearest-even)
- May raise spurious exceptions; does not maintain/set errno

SVML Command-Line Options

`-fimf-precision=value[:funclist]` - defines the accuracy (precision) for math library functions
value: **high** (error ≤ 1 ulp); **medium** (4 ulp; DEFAULT); **low** (11 accurate bits for single prec., 26 for double prec.)
funclist - optional list of one or more math library functions to which the attribute should be applied

`-fimf-accuracy-bits=bits[:funclist]` – defines the relative error, measured by the number of correct bits
bits - a positive, floating-point number

`-fimf-max-error=ulps[:funclist]` – max. allowable rel. error for math function results, including division and sqrt
ulps – the maximum allowable relative error the compiler should use

`-fimf-absolute-error=value[:funclist]` – defines the max. allowable absolute error for math library function results
value – max. allowable absolute error the compiler should use; the function may exceed the max. relative error setting

`-fimf-arch-consistency=value[:funclist]` – math functions produce consistent results on different implementations of the some architecture
value - true or false

`-fimf-force-dynamic-target[=funclist]` – use run-time dispatch in calls to math functions; off by default

`-fimf-use-svml=value[:funclist]` – use SVML rather than LIBM to implement math library functions; off by default
value - true or false

`-[no-]fast-transcendentals` – compiler may replace calls to transcendental functions with implementations that may be faster but less precise

`-fimf-domain-exclusion=classlist[:funclist]` – indicates the input arguments domain on which math functions must provide correct results
classlist – floating-point values you can exclude from the function domain: extremes, nans, infinities, denormals, zeros

SVML Usage Example

icc -S -std=c99 -O2 -fimf-use-svml=true -fimf-precision=low math_func.c

```
#include <math.h>

void math_func (int length, float *a, float *b, float *c,
        float *restrict x1)
{
   __assume_aligned(a, 16);
   __assume_aligned(b, 16);
   __assume_aligned(c, 16);
   __assume_aligned(x1, 16);

   for (int i=0; i<length; i++) {
      x1[i] = erff(b[i]*b[i]) - cosf(4*a[i]*c[i]);
   }

}
```

Kernel:

```
..B1.4:
      movups    (%rbp,%r15,4), %xmm0
      mulps     %xmm0, %xmm0
      call      *__svml_erff4_ep@GOTPCREL(%rip) # Low acc.
..B1.15:
      movaps    %xmm0, %xmm9
      movups    (%rbx,%r15,4), %xmm0
      mulps     %xmm8, %xmm0
      mulps     (%r12,%r15,4), %xmm0
      call      *__svml_cosf4_ep@GOTPCREL(%rip) # Low acc.
..B1.14:
      subps     %xmm0, %xmm9
      movups    %xmm9, (%r13,%r15,4)
      addq      $4, %r15
      cmpq      %r14, %r15
      jb        ..B1.4
```

Intel® Math Kernel Library Vector Math

Example: y(i) = $e^{x(i)}$ for *i =1 to n*

Vector Math

- Arithmetic: add, sub, …
- Trigonometric: sin, cos, …
- Hyperbolic: sinh, cosh, …
- Exponential: exp, exp2, …
- Log: log, log10, …
- Power: pow
- Root: sqrt, …
- Rounding: ceil, floor, …

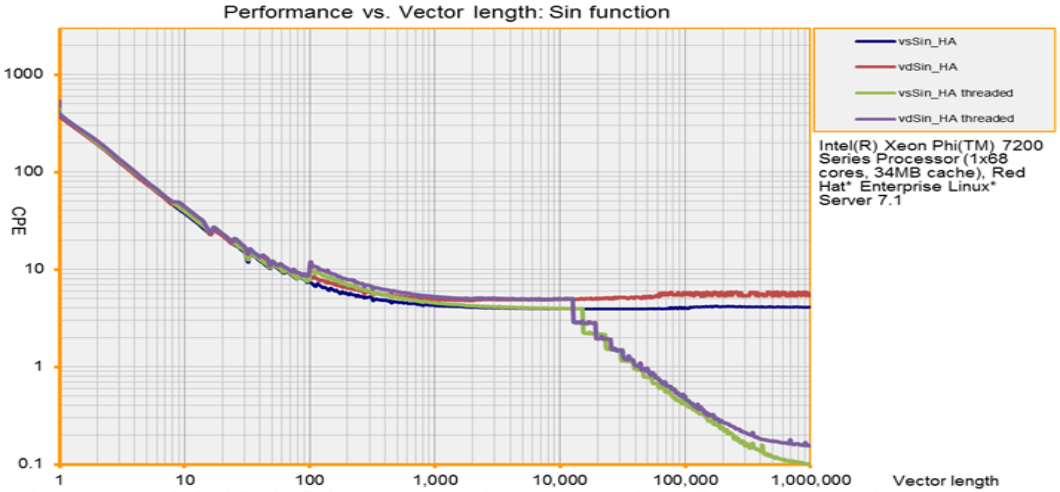
Performance-oriented transcendental math functions with vector API

- Arguments are pointers to memory arrays used as input/output vectors
- Vector length is passed as parameter
- Single precision, double precision; real and complex
- Maximum performance for long vectors
- Internal automatic threading, with OpenMP* or Intel® Threading Building Blocks(Intel® TBB)
- Multiple accuracy levels:
  - High (error < 1 ulp), aka HA (High Accuracy)
  - Medium (default; error < 4 ulp), aka LA (Low Accuracy)
  - Low (~half of the mantissa bits are correct), aka EP (Enhanced Performance)
- Uses a CPU-based internal dispatcher
- Advanced error handling

```
MKL VM API example: void vsExp (const MKL_INT n, const float* a, float* y);
```

Vector-based elementary functions allow developers to balance accuracy and performance

Intel® Math Kernel Library VM Threading Control



- Threading starts for lengths ≳ 16K
- Number of threads dynamically adjusted
- Number of threads and affinity manual control:
  - ✓ environment variables (OMP_NUM_THREADS, MKL_NUM_THREADS, MKL_DOMAIN_NUM_THREADS, KMP_AFFINITY)
  - ✓ service functions (omp_set_num_threads(), mkl_set_num_threads(), mkl_domain_set_num_threads())

OpenCL* Compiler Math Library Built-Ins

OpenCL* transcendental math built-ins with fixed vector length API and additional accuracy levels

- Math functions have OpenCL* standard vector data types as input/output, single and double precision
- Supported platforms (CPU): Intel® SSE4.2, Intel® AVX, Intel® AVX2, Intel® AVX512
- Various floating-point vector data type lengths: float1/2/4/8/16, double1/2/4/8/16
- Accuracy levels: generic, half, and native
  - Generic accuracy: varies from correctly rounded to 16 ulp-s, based on operation/function
  - Half accuracy: less than 8192 ulp (we use EP functions, with callouts)
  - Native accuracy: implementation-defined (we use EP functions without callouts)
- Function families
  - Converts (floating-point to integer, etc.)
  - Trigonometric (sin, cos, asin, acos, etc.)
  - Exp, Log, Power (exp, log, pow, etc.)
  - Hyperbolic (sinh, cosh, etc.)
  - Miscellaneous (floor, trunc, fmax, etc.)

OpenCL* API examples for math built-ins:

```
float4    exp(float arg);
float8    half_log (float8 arg);
float16   native_powr(float16 argX, float16 argY);
```

Black-Scholes Benchmark (Excerpt) – Will Implement w/ Compiler SVML and Intrinsics, OpenCL, MKL VM

- Stock option pricing with the Black-Scholes formula (based on solving PDEs)

```
for ( i = 0; i < nopt; i++ ) {
   d1[i]= (logf(s0[i]/x[i]) + (r + 0.5f*sig*sig) * t[i]) / (sig*sqrtf(t[i]));
   d2[i]= (logf(s0[i]/x[i]) + (r -  0.5f*sig*sig) * t[i]) / (sig*sqrtf(t[i]));
}
```

Ways to improve performance:

- Enable the auto-vectorizer of the Intel® Compiler
- Use intrinsics + direct SVML calls
- OpenCL* programming with vector types and math built-ins
- Call Intel® Math Kernel Library vector math functions (MKL VM)
- Use asm (least favorite solution!)

Black-Scholes Benchmark: Auto-Vectorization

```
#pragma vector always
#pragma ivdep
for ( i = 0; i < nopt; i++ )
{
   d1[i]= (logf(s0[i]/x[i]) + (r + 0.5f*sig*sig) * t[i]) / (sig*sqrtf(t[i]));
   d2[i]= (logf(s0[i]/x[i]) + (r -  0.5f*sig*sig) * t[i]) / (sig*sqrtf(t[i]));
}
```

Tell compiler to always vectorize this loop and to ignore possible vector dependencies

Pros:

- Minimal effort
- Rely on compiler optimizations
- Results in close-to-optimal performance

Cons:

- Limited ability for fine performance tuning

Black-Scholes Benchmark: Auto-Vectorization vs. Baseline

```
..B1.3:
      vmovss    (%rbp,%rbx,4), %xmm0
      vdivss    (%r12,%rbx,4), %xmm0, %xmm0
      call      logf
      ...
      vfmadd231ss 56(%rsp), %xmm0, %xmm1
      vsqrtss   %xmm0, %xmm0, %xmm0
      vmulss    72(%rsp), %xmm0, %xmm2
      vdivss    %xmm2, %xmm1, %xmm3
      ...
      vdivss    (%r12,%rbx,4), %xmm4, %xmm0
      vmovss    %xmm3, (%r14,%rbx,4)
      ...
      vmovss    %xmm3, (%r15,%rbx,4)
      incq      %rbx
      cmpq      48(%rsp), %rbx
jl      ..B1.3
```

Baseline:

- Scalar single precision SSE instructions used
- Non-optimized operations (div, sqrt)
- Calls to scalar LIBM math functions made

```
..B1.13:
      vmovups   (%r12,%r15,4), %ymm1
      vrcpps    %ymm1, %ymm2
      vfnmadd213ps %ymm8, %ymm2, %ymm1
      vmulps    %ymm2, %ymm1, %ymm3
      ...
      call      *__svml_logf8_l9@GOTPCREL(%rip)
      ...
      vrsqrtps  %ymm4, %ymm3
      vfmadd231ps %ymm12, %ymm4, %ymm0
      ...
      vrcpps    %ymm1, %ymm2
      vfnmadd213ps %ymm8, %ymm2, %ymm1
      vmulps    %ymm2, %ymm1, %ymm3
      ...
      vmovups   %ymm0, (%rdi,%r15,4)
      addq      $8, %r15
      cmpq      %r14, %r15
jb      ..B1.13
```

Auto-vectorized:

- Vector packed single precision SSE instructions are used
- Optimized operations (div, sqrt) are inlined
- Calls made to vector SVML math functions

Black-Scholes Benchmark: Intrinsics + Direct SVML Calls

```
for ( i = 0; i < nopt; i+=8 ) {
    __m256 t1 = _mm256_div_ps(s0,x);
    __m256 t2 = _mm256_log_ps(t1);
    __m256 t3 = _mm256_invsqrt_ps(t);
    __m256 t4 = _mm256_add_ps(r, sig2over2);
    __m256 d1 = _mm256_fmadd_ps(t4, t, t2);
    d1 = _m256_mul_ps(d1, invsig);
    d1 = _m256_mul_ps(d1, t3);
     ...
}
```

Use compiler intrinsics for SIMD types + direct calls to SVML functions

Pros:

- Fine performance tuning ability

Cons:

- Significant design/support effort

Black-Scholes Benchmark: OpenCL* Code

```
{
   float8 t1 = s0 / x;
   float8 t2 = log(t1);
   float8 t3 = 1 / half_sqrt(t);
   float8 t4 = r + sig2over2;
   float8 d1 = (t4*t + t2)*invsig*t3;
  ...
 }
```

Use vector data types + math built-in functions

Pros:

- Direct access to additional accuracy levels and vector length control
- High level programming language

Cons:

- Less fine performance tuning control

Black-Scholes Benchmark Using Intel® MKL Vector Math

```
float t1[N], t2[N], t3[N];
vsDiv(N, s0, x, t1);
vsLn(N, t1, t2);
vsInvSqrt(N, t, t3);

#pragma vector always
#pragma ivdep
for ( i = 0; i < N; i++ )
{
   d1[i]= (t2[i] + (r + 0.5f*sig*sig) * t[i]) * invsig * t3[i];
   d2[i]= (t2[i] + (r - 0.5f*sig*sig) * t[i]) * invsig * t3[i];
}
```

Call MKL Vector Math (VM) functions on memory buffers of length N, where N is large enough (>1000)

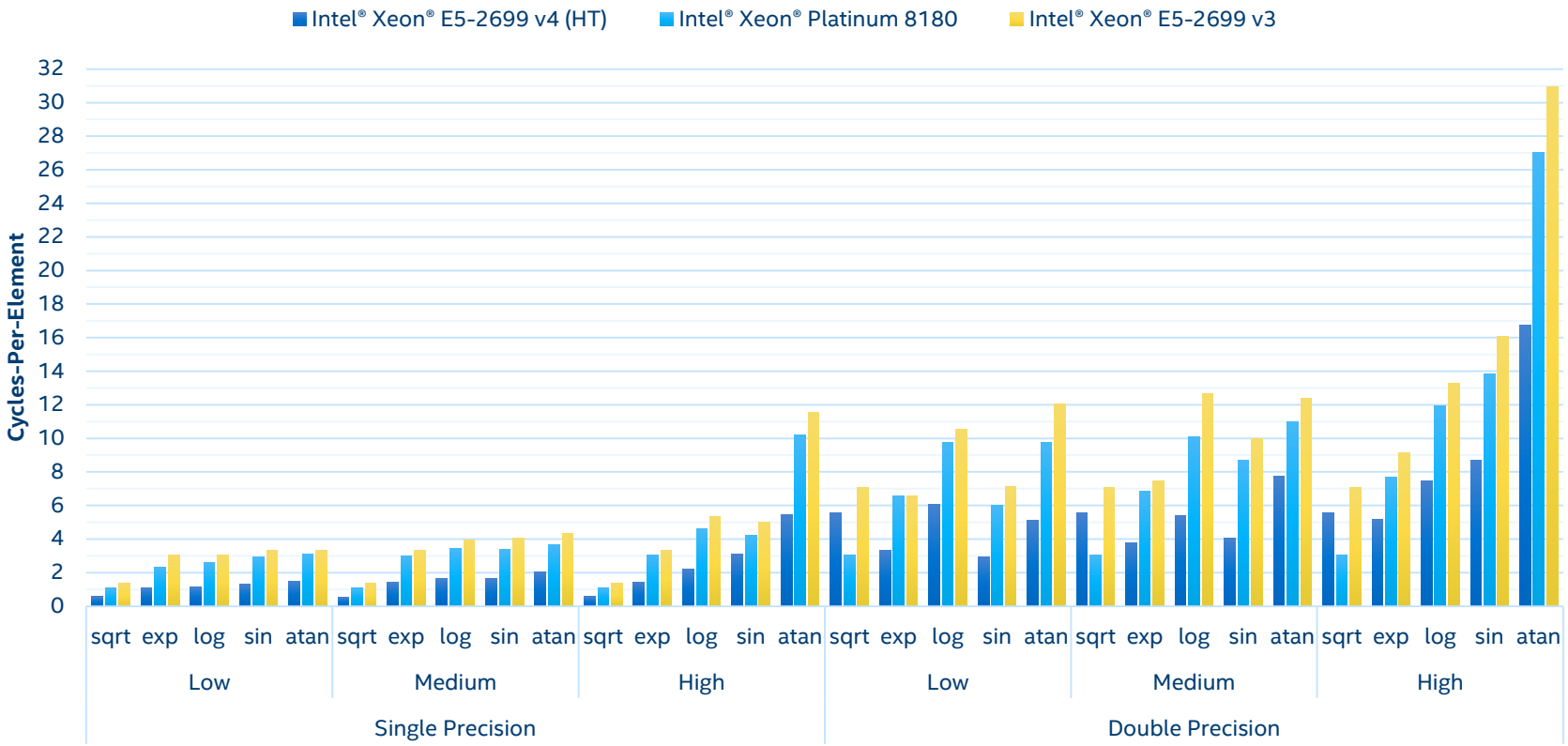
Pros:

- High performance on large vectors

Cons:

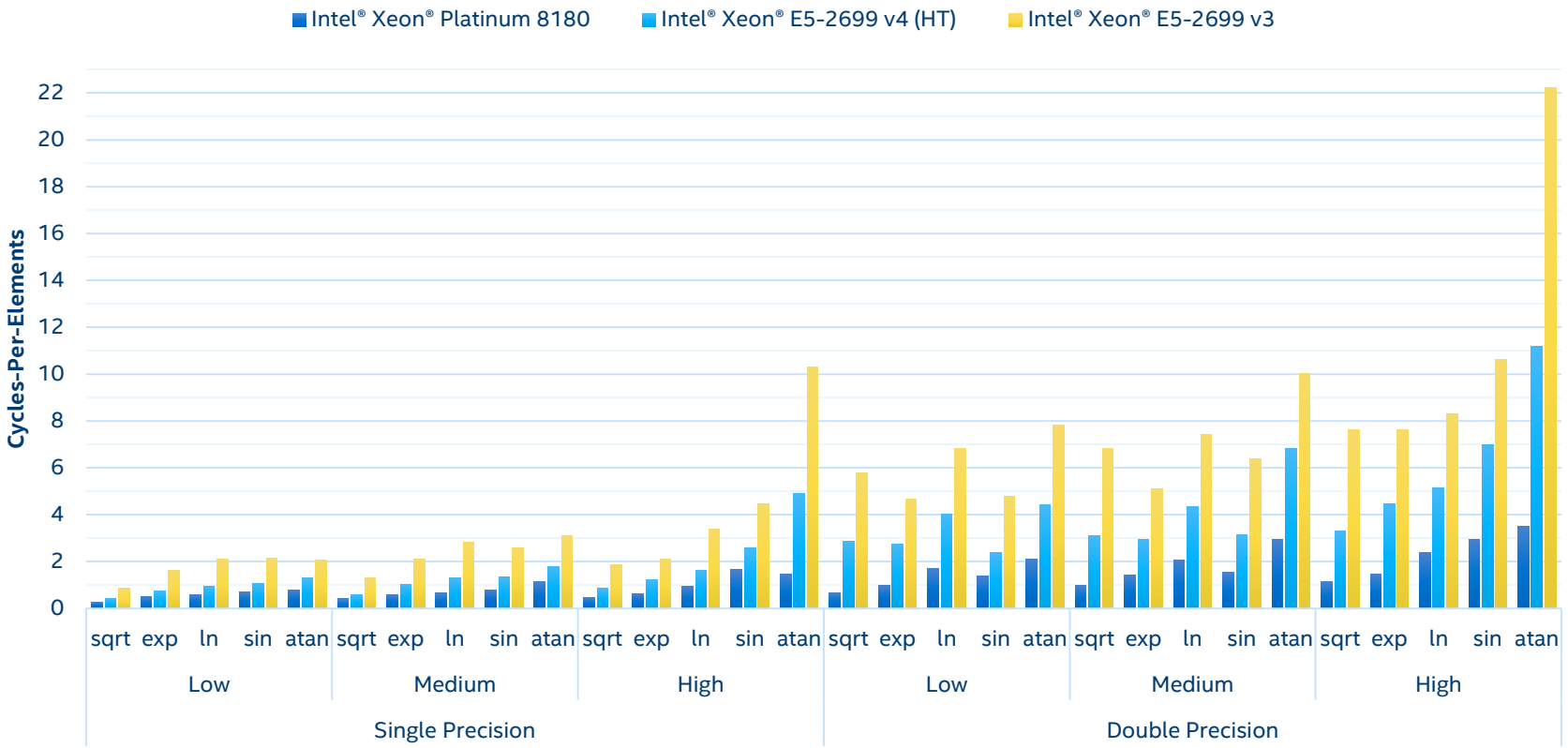
- Extra memory consumption for intermediate buffers

Intel® Short Vector Math Library Throughput Performance



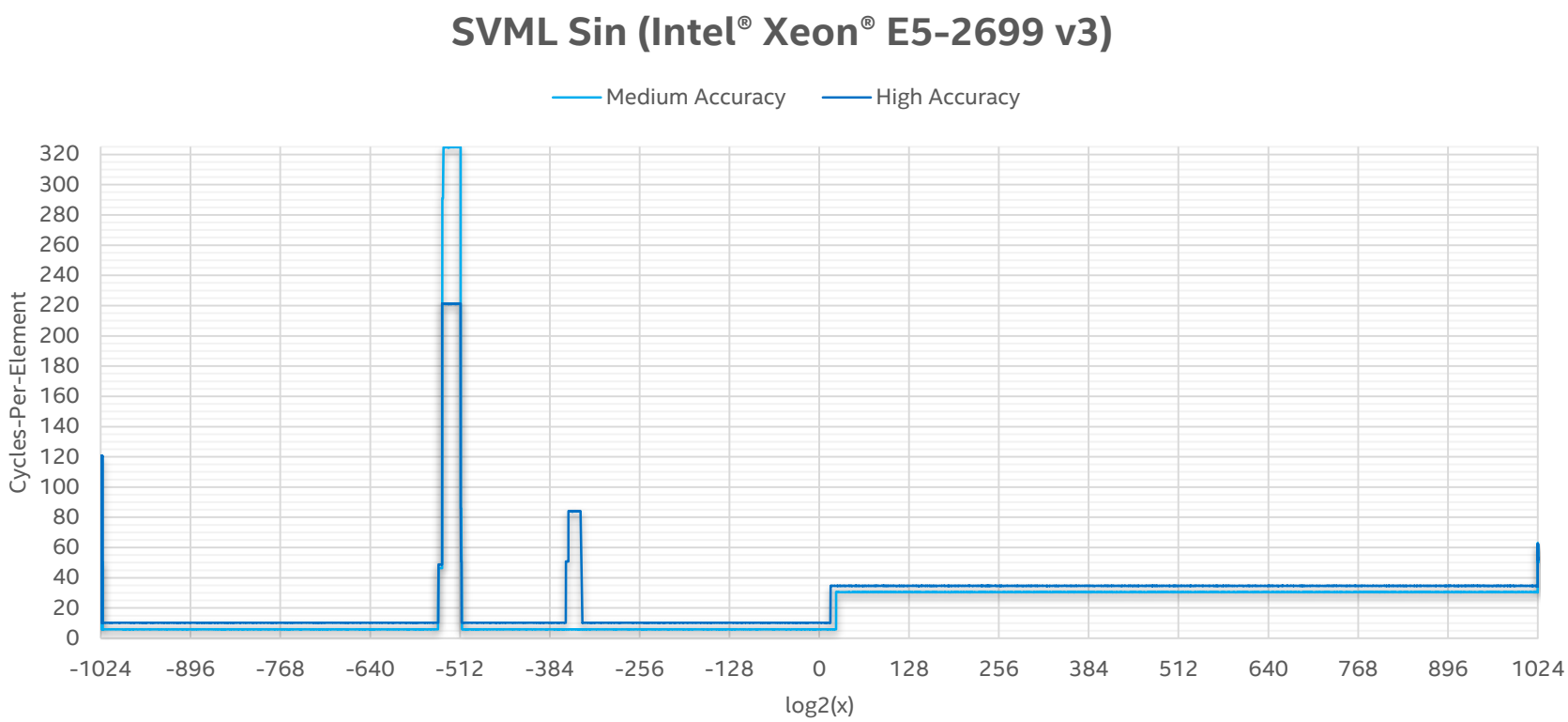
Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

Intel® Math Kernel Library Vector Math Performance (N = 1024)



Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

Intel® Short Vector Math Library (SVML) Performance



References

- Intel® 64 and IA-32 Architectures Software Developer Manuals - https://software.intel.com/en-us/articles/intel-sdm
- Intel® Advanced Vector Extensions – Intel® AVX-512 https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html, https://software.intel.com/en-us/isa-extensions/intel-avx
- Intel® C++ Compiler 18.0 Developer Guide and Reference - https://software.intel.com/en-us/cpp-compiler-18.0-developer-guide-and-reference
- Intel® Math Kernel Library - https://software.intel.com/en-us/mkl
- Intel® SDK for OpenCL™ Applications - https://software.intel.com/en-us/intel-opencl

NOTICES AND DISCLAIMERS

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks .

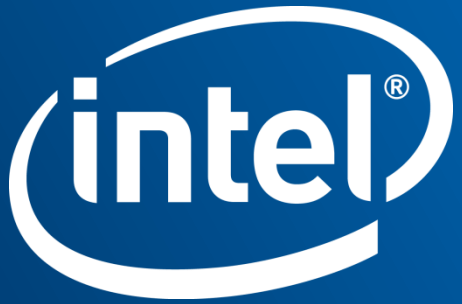
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/benchmarks .

Intel® Advanced Vector Extensions (Intel® AVX)* provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.
Intel, the Intel logo, and Intel Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as property of others.


intel®