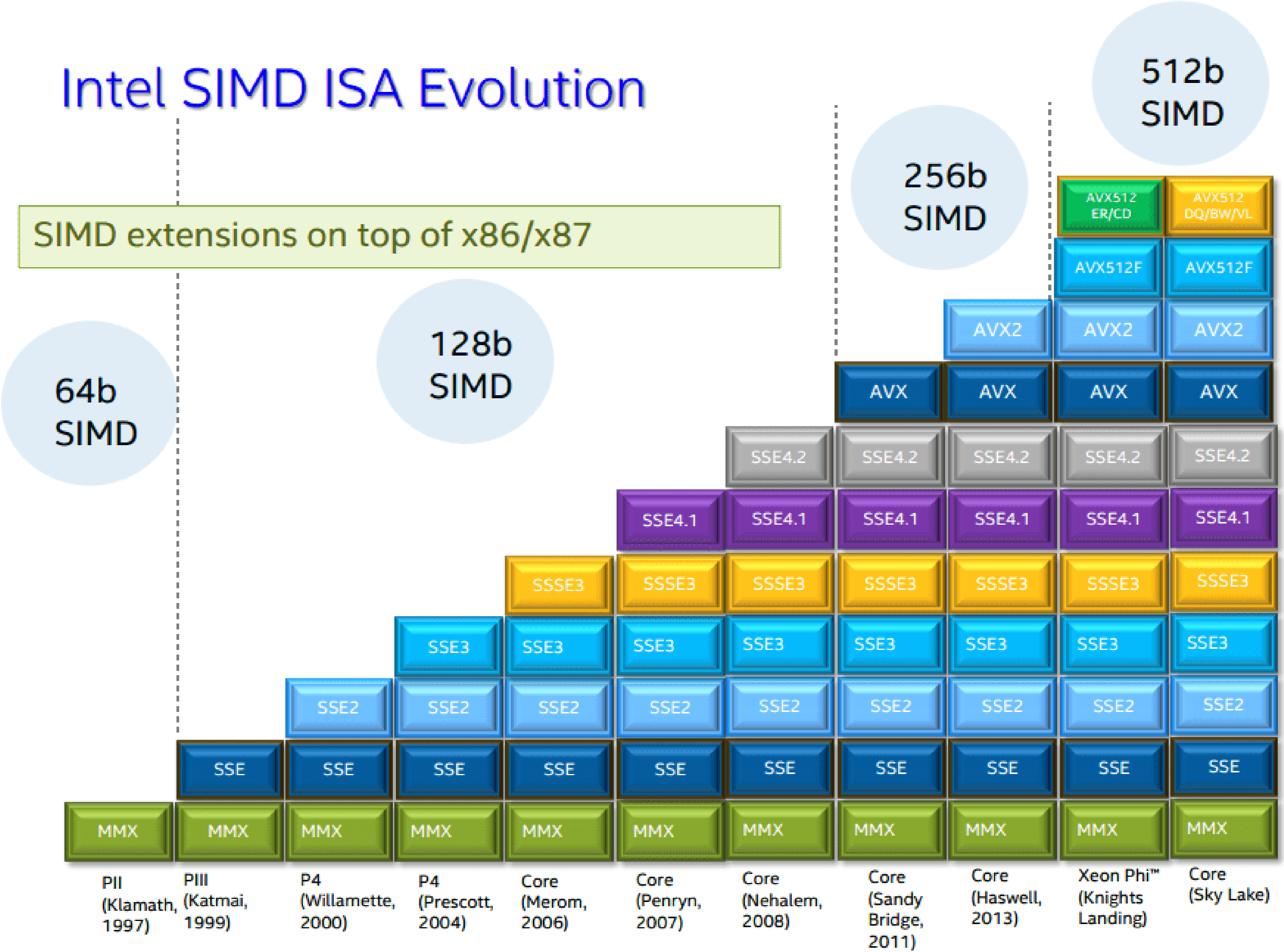
SIMD and AVX512

Based on the slides by Chad Jarvis
Simula Research Laboratory

History of Intel x86/x87 SIMD



https://www.researchgate.net/profile/Mostafa_Soliman3/publication/288993171_State-of-the-Art_in_Processor_Architecture/links/56881ca308ae19758398eb08/State-of-the-Art-in-Processor-Architecture.pdf

AVX512 with intrinsics

The preferred method for low programming is using ***intrinsics*** instead of assembly. This is because intrinsics are much more convenient (except for their names).

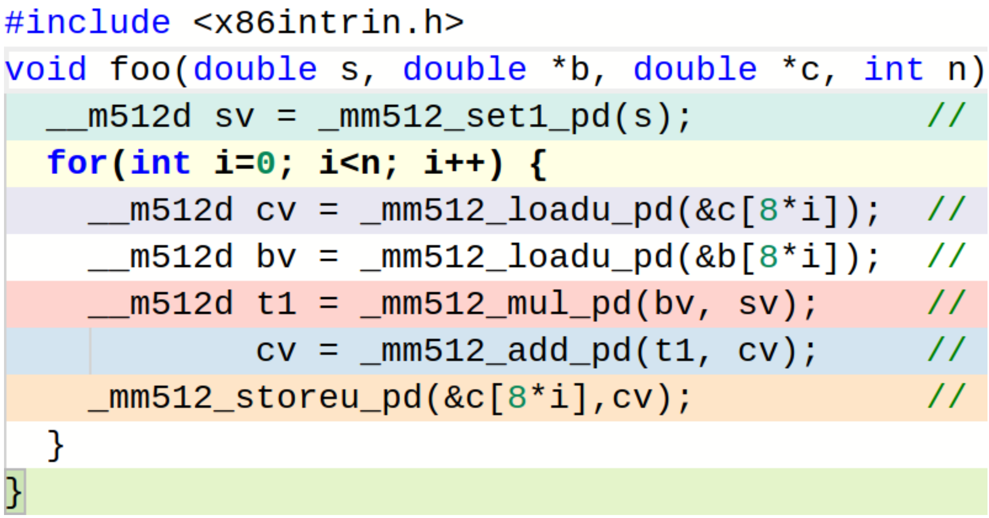
```
void foo(double s, double *b, double *c, int n) {
  __m512d sv = _mm512_set1_pd(s);             // broadcast s to all 8 values
  for(int i=0; i<n/8; i++) {
    __m512d cv = _mm512_loadu_pd(&c[8*i]);    // load 8 double from c
    __m512d bv = _mm512_loadu_pd(&b[8*i]);    // load 8 double from b
    __m512d t1 = _mm512_mul_pd(bv, sv);       // sv*bv
            cv = _mm512_add_pd(t1, cv);       // cv = sv*av + cv
    _mm512_storeu_pd(&c[8*i],cv);             // write cv to c array
  }
}
```

Notice that the loop goes over n/8 elements. In principle this loop can be eight times faster than the scalar loop in the function foo.

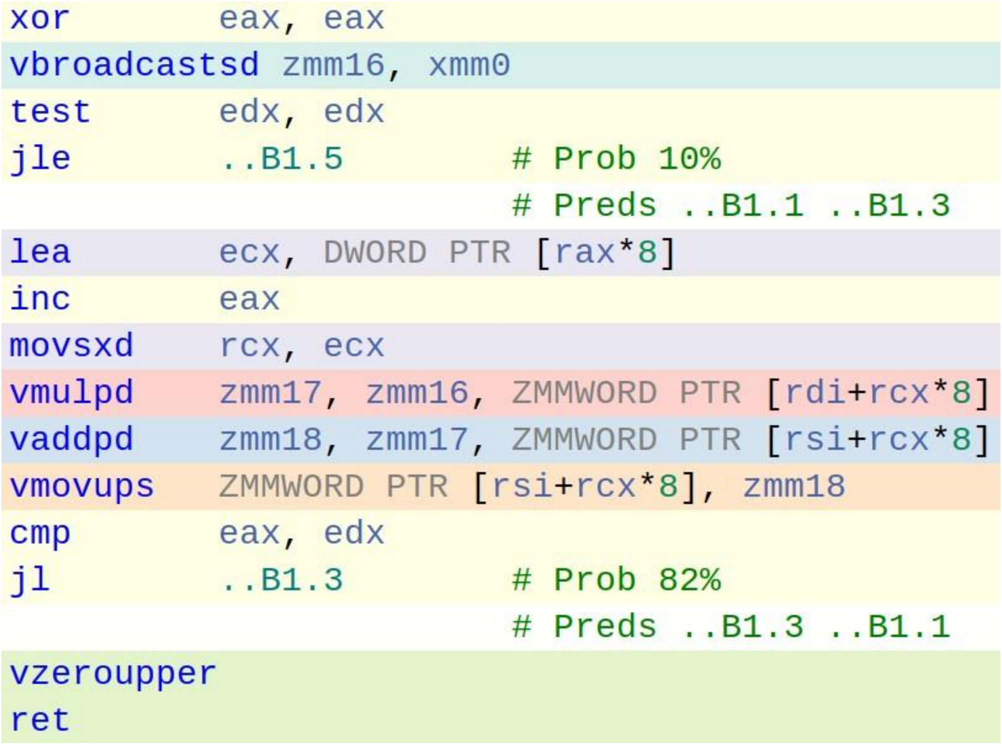
10 times faster than gemm

Intrinsics and assembly

Use GodBolt (https://godbolt.org/) to see the assembly for GCC, ICC, Clang, and MSVC



```
#include <x86intrin.h>
void foo(double s, double *b, double *c, int n)
  __m512d sv = _mm512_set1_pd(s);           //
  for(int i=0; i<n; i++) {
    __m512d cv = _mm512_loadu_pd(&c[8*i]);  //
    __m512d bv = _mm512_loadu_pd(&b[8*i]);  //
    __m512d t1 = _mm512_mul_pd(bv, sv);     //
            cv = _mm512_add_pd(t1, cv);     //
    _mm512_storeu_pd(&c[8*i],cv);           //
  }
}
```



```
xor       eax, eax
vbroadcastsd zmm16, xmm0
test      edx, edx
jle       ..B1.5        # Prob 10%
                        # Preds ..B1.1 ..B1.3
lea       ecx, DWORD PTR [rax*8]
inc       eax
movsxd    rcx, ecx
vmulpd    zmm17, zmm16, ZMMWORD PTR [rdi+rcx*8]
vaddpd    zmm18, zmm17, ZMMWORD PTR [rsi+rcx*8]
vmovups   ZMMWORD PTR [rsi+rcx*8], zmm18
cmp       eax, edx
jl        ..B1.3        # Prob 82%
                        # Preds ..B1.3 ..B1.1
vzeroupper
ret
```

Intrinsics

- #include <x86intrin.h> for GCC, ICC, and Clang. For MSVC see https://stackoverflow.com/questions/11228855/header-files-for-x86-simd-intrinsics

	Data type	Intrinsic prefix
SSE	__m128 (float) __m128d (double) __m128i (int)	_mm_
AVX	__m256 (float) __m256d (double) __m256i (int)	_mm256_
AVX512	__m512 (float) __m512d (double) __m512i (int)	_mm512_

Advantages of Intrinsics

Advantages of intrinsics over assembly:

- You don't have to worry about 32-bit and 64-bit mode.
  - 32-bit and 64-bit assembly syntax are incompatible.
- You don't need to worry about registers and register spilling.
  - There are a finite number of logical registers, in assembly if you use them all, and need more, then you have to manage this manually.
- No need to worry about AT&T and Intel assembly syntax.
  - There are two competing dialects of assembly (like bokmål and nynosk).
- No need to worry about function calling conversions.
  - 32-bit and 64-bit and Windows, MacOS, and Linux all use different function calling conventions which you have to adhere to in assembly.
- The compiler can optimize intrinsics further which it won't do with inline assembly.
- Intrinsics are compatible (mostly) with the four most popular C/C++ compilers: GCC, ICC, Clang, MSVC.

Disadvantages of Intrinsics

The main disadvantage of intrinsics are:

- Complicated Names: e.g. __m512d v1 = _mm512_fmadd_pd(sv, bv, cv)
  Only compatible with x86 hardware.
- If you want exact control you still need assembly.
- Some hardware features can only be accessed with assembly.

Fused Multiply Add (FMA)

fma(a,b,c) does a*b + c in one operation
This can both improve performance and accuracy.

Without FMA
p = round(a*b)
s = round(p + c): two operations, round twice

With FMA
s = round(a*b + c): one operation, round once

FMA with intrinsics

```
void foo_intrin_fma(double s, double *b, double *c, int n) {
  __m512d sv = _mm512_set1_pd(s);             // broadcast s to all 8 values
  for(int i=0; i<n/8; i++) {
    __m512d cv = _mm512_loadu_pd(&c[8*i]);   // load 8 double from c
    __m512d bv = _mm512_loadu_pd(&b[8*i]);   // load 8 double from b
            cv = _mm512_fmadd_pd(sv, bv, cv); // cv = sv*bv + cv
    _mm512_storeu_pd(&c[8*i], cv);            // write cv to c array
  }
}
```

Modern compilers will likely convert

```
        t1 = _mm512_mul_pd(bv, sv);       // sv*bv
        cv = _mm512_add_pd(t1, cv);       // cv = sv*av + cv
```

to

```
        cv = _mm512_fmadd_pd(sv, bv, cv); // cv = sv*bv + cv
```

FMA4: four operands e.g. d = a*b + c

FMA3: three operands e.g. c = a*b + c

Intel proposed FMA4, AMD implemented it first, Intel came out with FMA3 instead.

The Vector Class Library (VCL)

C++ library for SIMD operations

https://www.agner.org/optimize/#vectorclass

```
void foo_VCL(double s, double *b, double *c, int n) {
  Vec8d sv = s;                       // broadcast s to all 8 values
  for(int i=0; i<n/8; i++) {
    Vec8d cv = Vec8d().load(&c[8*i]); // load 8 double from c
    Vec8d bv = Vec8d().load(&b[8*i]); // load 8 double from a
    cv = sv*bv + cv;
    cv.store(&c[8*i]);                // write cv to c array
  }
}
```

In most cases you will get the same performance as using intrinsics.

Peak FLOPS for GEMM

Instruction SET		FLOP /cycle
Intel SSE2 Core2 (2006)	(2-wide MUL)+(2-wide ADD)	4
AMD AVX Bulldozer (2011)	(2 FLOP/FMA)*(4-wide FMA)	8
Intel AVX (2011)	(4-wide MUL)+(4-wide ADD)	8
AMD AVX ZEN (2017)	(2 FLOP/FMA)*(4-wide FMA)	8
Intel AVX2 (2013)	(2 FLOP/FMA)((4-wide FMA)+(4-wide FMA))	16
AMD AVX2 ZEN2 (2019?)	(2 FLOP/FMA)((4-wide FMA)+(4-wide FMA))	16
AVX512 (single issue)	(2 FLOP/FMA)*(8-wide FMA)	16
AVX512 (dual issue)	(2 FLOP/FMA)((8-wide FMA)+(8-wide FMA))	32

Peak double floating point flops, for single floating point double the FLOP/cycle

Performance tests for a 1024x1024

1024x1024 matrix: floating point operations = $2*n^3$ = 2.15 GFLOP

Intel Xeon 6142 CPU (duel issue AVX512) @ 3.5 GHz and AVX512
Peak FLOPS for a single core = 32*3.5GHZ = 112 GFLOPS

n=1024	Time	GFLOPS	GFLOPS/peak
gemm1	4.0 s	0.5	0.3 %
gemm2	0.7 s	3.0	2.7 %
gemm2 with foo_intrin	0.35 s	6.0	5.3 %
gemm2 with foo_intrin_fma	0.35 s	6.0	5.3%

The code is memory bandwidth bound. To get closer to the peak flops
(I have achieved over 70% with more advanced code) you have you use loop tiling.

Frequency scaling and dark silicon

Intel Xeon 6142 CPU 16 cores
One of the big problems with AVX and especially AVX512 is that they use so much power that the processor can’t operate at its thermal design power (TDP).

Instruction Set	1 or 2 cores	16 cores
SSE	3.7 GHz	3.3 GHz
AVX	3.6 GHz	2.9 GHz
AVX512	3.5 GHz	2.2 GHz

It’s not possible to use all the cores at max frequency with AVX or AVX2.
Some system administrators disable AVX512 on servers because code with AVX512 from a single user can greatly effect the performance of another user’s code without AVX512.

Dark silicon means that not all the logic (silicon) can be enabled at the TDP and so must be disabled or clocked down.

Operators in C

- Arithmetic
  a + b, a -b, a*b, a/b, a%b
- Bitwise
  a | b, a & b, a ^ b, ~a
- Bit shift
  a << b, a >> b (signed), a >> b (unsigned)
- Logical operators
  a && b, a || b, !a
- Comparison operators
  a == b, a != b, a < b, a <= b, a > b, a >= b
- Tertiary operator
  x = a ? b : c
- Special functions:
  sqrt(x), abs(x), fma(a,b,c), ceil(x), floor(x)

Arithmetic operators

	char	short	int	int64_t	float	double
+ (-)	SSE2	SSE2	SSE2	SSE2	SSE	SSE2
*	NA	SSE2	SSE4.1	AVX512DQ	SSE	SSE2
/	NA	NA	NA	NA	SSE	SSE2
%	NA	NA	NA	NA		

- No 8-bit multiplication (can be emulated with 16-bit multiplication)
- No integer division with x86 SIMD
- Floating point division is slow and only uses one port
  (internally it may only be 128-bit or 256-bit wide as well).
- The AVX512DQ 64-bit * 64-bit to 64-bit instruction is still slow.

Fast division for constant divisors

Calculate r = a/b where b is a constant

With floating point we precompute (at compile time or outside of the main loop) the inverse ib = 1.0/b.

```
r = ib*a
```

Floating point division with constant divisors becomes multiplication

With integers the inverse is more complicated

```
  ib,n = get_magic_numbers(b);
```

```
r = ib*a >> n
```

Integer division with constant divisors becomes multiplication and a bit-shift

Fast Division Examples

Dividing integers by a power of two can be done with a bit shift which is very fast.

- x/3 = x*1431655766/2^32

  27*1431655766/2^32 = 3

- x/1000 = x*274877907/2^38

  10000*274877907/2^32 = 10

- x/314159 = x*895963435/2

  7*314159*895963435/2^48 = 7

Bitwise operators

	char	short	int	int64_t	float	double
^ (XOR)	SSE2	SSE2	SSE2	SSE2	SSE	SSE2
| (OR) & (AND)	SSE2	SSE2	SSE2	SSE2	SSE	SSE2
>> unsigned >>	NA	SSE2	SSE2	AVX512 SSE2		
<<		SSE2	SSE2	SSE2		
~ (NOT)	NA	NA	NA	NA	NA	NA

- Signed integers use arithmetic shift which shifts in the sign bit for >>
- Signed and unsigned integers use logical shift which shifts in zeros.
- ~x = (x ^ -1)

Comparison operators

	char	short	int	int64_t	float	double
==	SSE2	SSE2	SSE2	SSE4.1	SSE	SSE2
!=	AVX512BW XOP	AVX512BW XOP	AVX512F XOP	AVX512F XOP	SSE	SSE2
< (>)	NA	SSE2	SSE2	SSE4.2	SSE	SSE2
<= (>=)	AVX512BW or XOP	AVX512BW XOP	AV512F XOP	AVX512F XOP	SSE	SSE2

- (a != b) = ~(a == b) = (a == b) ^ -1

Other Vertical Operators

	char	short	int	int64_t	float	double
? :	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE4.1
fma3 fma4	NA	NA	NA	NA *	AVX2 XOP	AVX2 XOP
min (max)	SSE4.1	SSE2	SSE4.1	AVX512F	SSE	SSE2
sqrt	NA	NA	NA	NA	SSE	SSE2
abs	SSSE3	SSSE3	SSSE3	AVX512F	AVX512F	AVX512F
floor/ceil/ round					SSE4.1	SSE4.1

* AVX512IFMA52 has FMA for 52-bit integers

Horizontal Operators

Shuffle/permute/swizzle elements within a vector

- e.g. v(1, 2, 3, 4) to v(4, 3, 2, 1)

Add elements within a vector

- e.g. v(1, 2, 3, 4) to v(1 + 2, 3 + 4, 0, 0)

There are a massive number of ways to permute elements within vectors e.g.

- permute(v1, int): where int is a compile time integer
- permute(v1, var): where var is a runtime time var
- permute(v1, v2, int), permute elements within vectors with a compile time integer

	char	short	int	int64_t	float	double
hadd (hsub)	SSE4.1	SSE4.1	SSE4.1	SSE4.1	SSE3	SSE3
permute(v1,var)	SSSE3	SSSE3	SSSE3 or AVX512F	SSSE3	AVX	AVX

Horizontal addition example

A reduction adds each element of an array and returns the sum

```
double reduction(double *a, int n) {
  double s = 0;
  for(int i=0; i<n; i++) {
    s + = a[i];
  }
  return s;
}
```

```
double reduction_VCL(double *a, int n) {
  Vec8d s1 = 0;
  for(int i=0; i<n/8; i++) {
    s1 += Vec8d().load(&a[8*i]);   //eight partial sums
  }
  return horizontal_add(s1);       //add the eight partial sums
}
```

Horizontal add example

```
double reduction_avx512(double *a, int n) {
  __m512d s1 = _mm512_setzero_pd();
  for(int i=0; i<n/8; i++) {
    __m512d av = _mm512_loadu_pd(&a[8*i]);
    s1 = _mm512_add_pd(cv,s1);                // sum = av + sum
  }
                                              // s1 =  1  2  3  4  5  6  7  8
  __m256d l1 = _mm512_castpd512_pd256(s1);    // l1 =  1  2  3  4
  __m256d h1 = _mm512_extractf64x4_pd(s1,1);  // h1 =  5  6  7  8
  __m256d s2 = _mm256_add_pd(h1,l1);          // s2 =  6  8 10 12
  __m256d s3 = _mm256_hadd_pd(s2, s2);        // s3 = 14 14 22 22
  __m128d l3 = _mm256_castpd256_pd128(s3);    // l3 = 14 14
  __m128d h3 = _mm256_extractf128_pd(s3,1);   // h3 = 22 22
  __m128d s4  = _mm_add_sd(l3,h3);            // s4 = 36 36
  return _mm_cvtsd_f64(s4);                   // return 36
}
//hadd_pd(a,b) =(a0+a1, b0+b1, a2+a3, b2,b3)
```

We can reduce a vector of width w in ln2(w) sums.
For 8 doubles it takes 3 sums.
For 16 floats it takes 4 sums.

Transposing a 4x4 float matrix

```
                               // row0  1  2  3  4
                               // row1  5  6  7  8
                               // row2  9 10 11 12
                               // row3 13 14 15 16

tmp0 = _mm_unpacklo_ps(row0, row1); //  1  5  2  6
tmp2 = _mm_unpacklo_ps(row2, row3); //  9 13 10 14
tmp1 = _mm_unpackhi_ps(row0, row1); //  3  7  4  8
tmp3 = _mm_unpackhi_ps(row2, row3); // 11 15 12 16

row0 = _mm_movelh_ps(tmp0, tmp2);   //  1  5  9 13
row1 = _mm_movehl_ps(tmp2, tmp0);   //  2  6 10 14
row2 = _mm_movelh_ps(tmp1, tmp3);   //  3  7 11 15
row3 = _mm_movehl_ps(tmp3, tmp1);   //  4  8 12 16
```

Scalar operations to transpose a matrix go as 2*n*(n-1) but only n*ln2(n) with SIMD

nxn matrix	4x4	8x8	16x16
SIMD ops	8	24	64
SIMD ops + R/W	16	40	96
Scalar ops + r/w	24	120	480

Gather and Scatter

Intel has created micro-code which essential implements the following

```
gathern(double *a, double *b, int *idx) {
  for(int i=0; i<n; i++) b[i] = a[idx[i]];
}

scattern(double *a, double *b, int *idx) {
  for(int i=0; i<n; i++) b[idx[i]] = a[i];
}
```

Where n is eight (four) with AVX512 (AVX2)

	char	short	int	int64_t	float	double
gather	NA	NA	AVX2	AVX2	AVX2	AVX2
scatter	NA	NA	AVX512F	AVX512F	AVX512F	AVX512F

Gather and scatter example

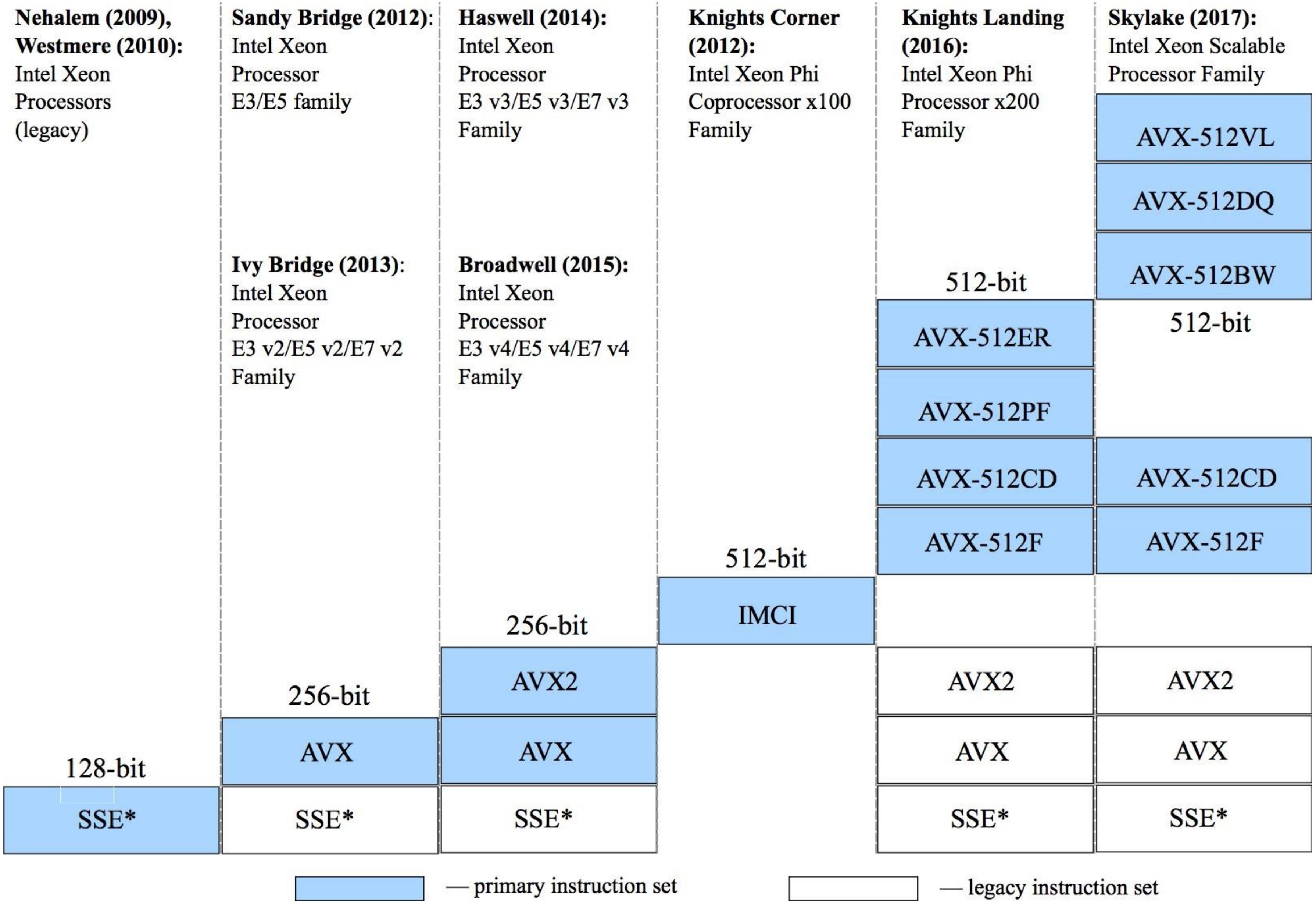
```
double  a[16] = {0, -1, 2, -3, 4, -5, 6, -7, 8, -9, 10, -11, 12, -13, 14, -15};
int idx[8] = {1, 3, 5, 7, 9, 11, 13, 15};
double b[8];

gather(a, b, idx); // b = {-1, -3, -5, -7, -9, -11, -13, -15}

memset(a, 0, 16*sizeof(double));
scatter(a, b, idx);
//a = {0, -1, 0, -3, 0, -5, 0, -7, 0, -9, 0, -11, 0, -13, 0, -15};
```

```
    __m256i vidx = _mm256_loadu_si256((__m256i*)idx);
    __m512d bv   = _mm512_i32gather_pd (vidx, a, 8);
                   _mm512_storeu_pd(b, bv);

    memset(a, 0, 16*sizeof(double));
    _mm512_i32scatter_pd(a, idx, bv, 8);
```



Nehalem (2009), Westmere (2010): Intel Xeon Processors (legacy)

128-bit

SSE*

Sandy Bridge (2012): Intel Xeon Processor E3/E5 family

Ivy Bridge (2013): Intel Xeon Processor E3 v2/E5 v2/E7 v2 Family

256-bit

AVX

SSE*

Haswell (2014): Intel Xeon Processor E3 v3/E5 v3/E7 v3 Family

Broadwell (2015): Intel Xeon Processor E3 v4/E5 v4/E7 v4 Family

256-bit

AVX2

AVX

SSE*

Knights Corner (2012): Intel Xeon Phi Coprocessor x100 Family

512-bit

IMCI

Knights Landing (2016): Intel Xeon Phi Processor x200 Family

512-bit

AVX-512ER

AVX-512PF

AVX-512CD

AVX-512F

AVX2

AVX

SSE*

Skylake (2017): Intel Xeon Scalable Processor Family

AVX-512VL

AVX-512DQ

AVX-512BW

512-bit

AVX-512CD

AVX-512F

AVX2

AVX

SSE*

🟦 — primary instruction set

□ — legacy instruction set

Example of AVX512 in Github

- https://github.com/jeffamstutz/tsimd