Intel SIMD

Chris Phillips| LBA Lead Scientist

November 2014

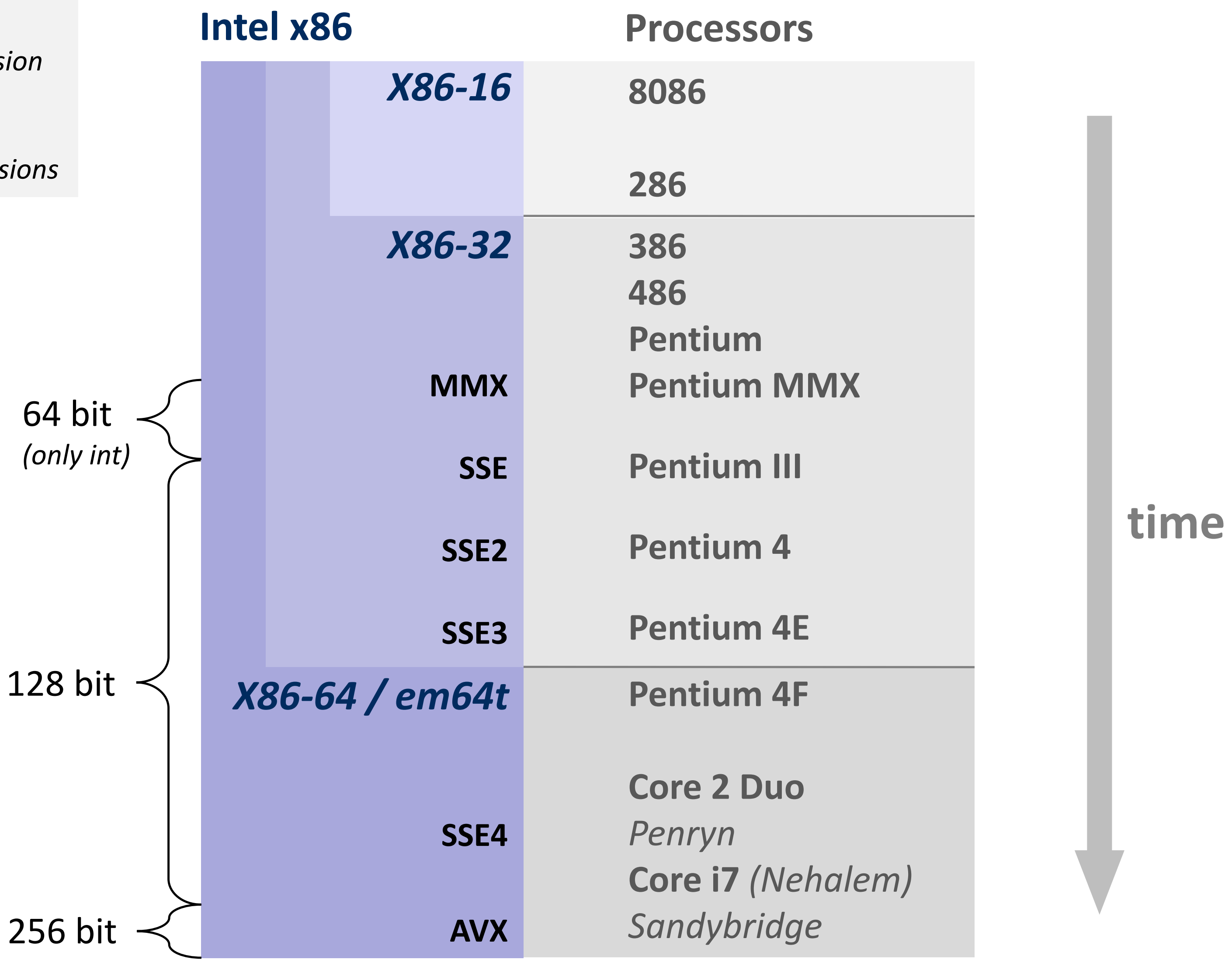
SIMD

- Single Instruction Multiple Data
- Vector extensions for x86 processors
- Parallel operations
- More registers than regular X86
- MXX, SSE, SSE2..4, AVX, AVX2….

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

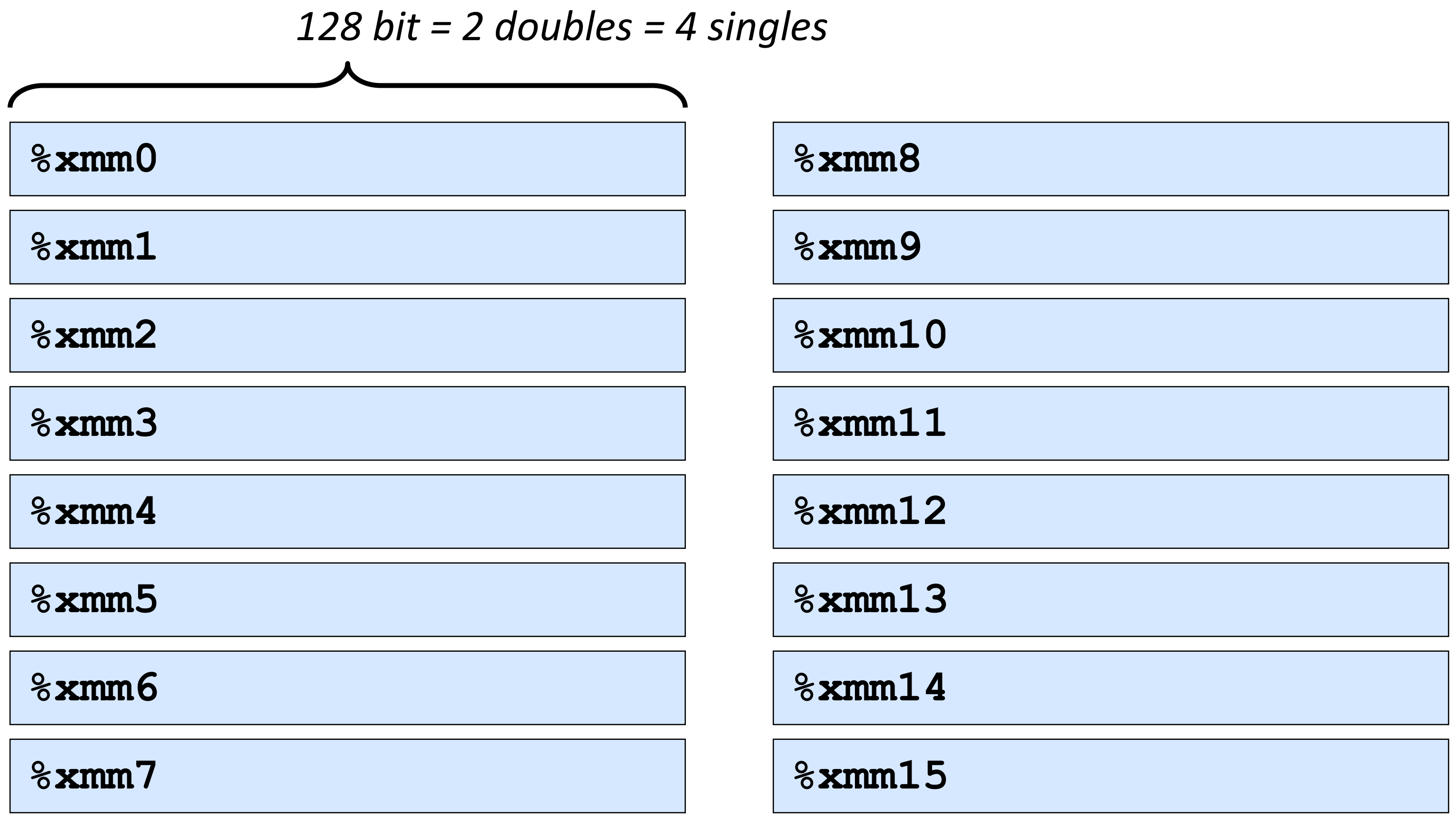
AVX:
Advanced vector extensions



Intel x86	**Processors**
X86-16	**8086**
	286
X86-32	**386**
	486
	Pentium
MMX	**Pentium MMX**
SSE	**Pentium III**
SSE2	**Pentium 4**
SSE3	**Pentium 4E**
X86-64 / em64t	**Pentium 4F**
	Core 2 Duo
SSE4	*Penryn*
	Core i7 *(Nehalem)*
AVX	*Sandybridge*

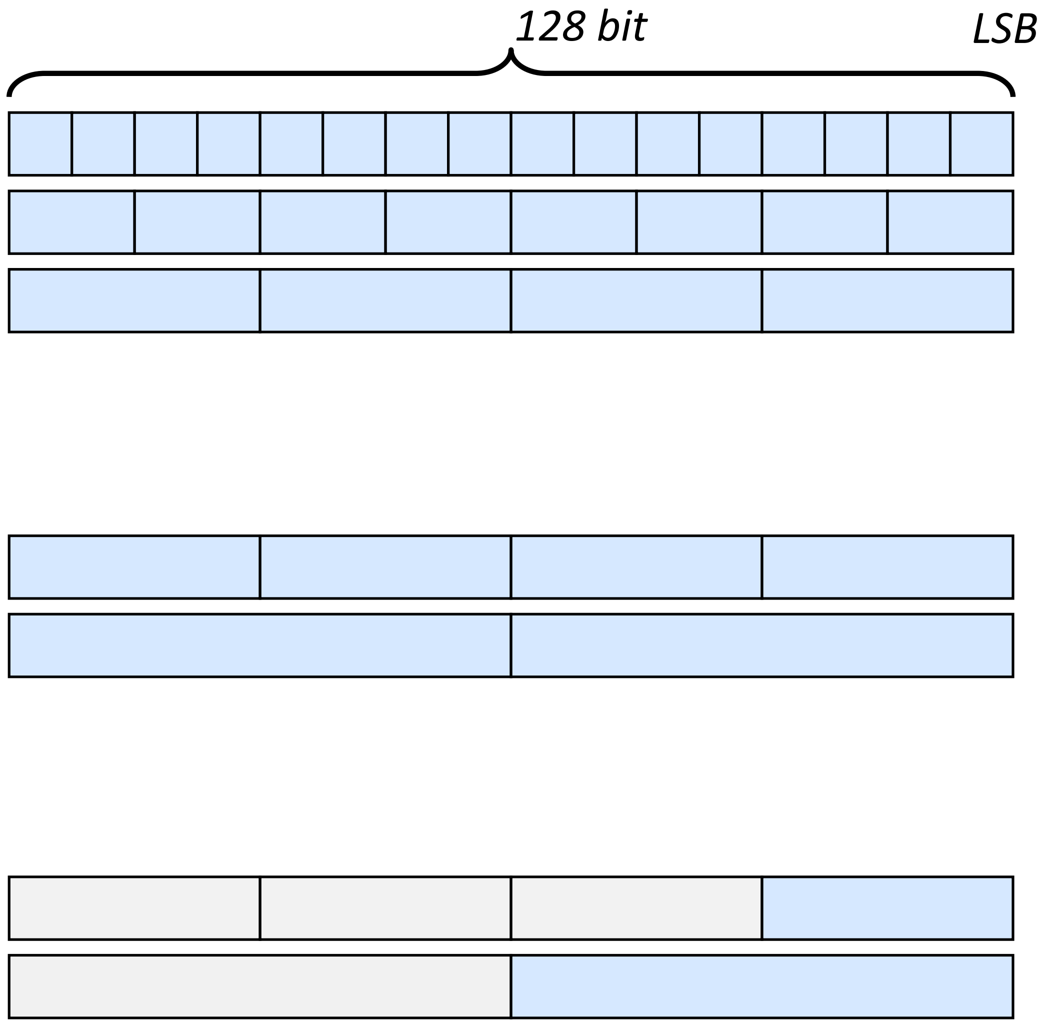
Core 2

- Has SSE3
- 16 SSE registers



SSE3 Registers

- **Different data types and associated instructions**
- **Integer vectors:**
  - 16-way byte
  - 8-way 2 bytes
  - 4-way 4 bytes
- **Floating point vectors:**
  - 4-way single (since SSE)
  - 2-way double (since SSE2)
- **Floating point scalars:**
  - single (since SSE)
  - double (since SSE2)



AVX

- Extends registers to 256bits
- AVX2 needed for integer operations

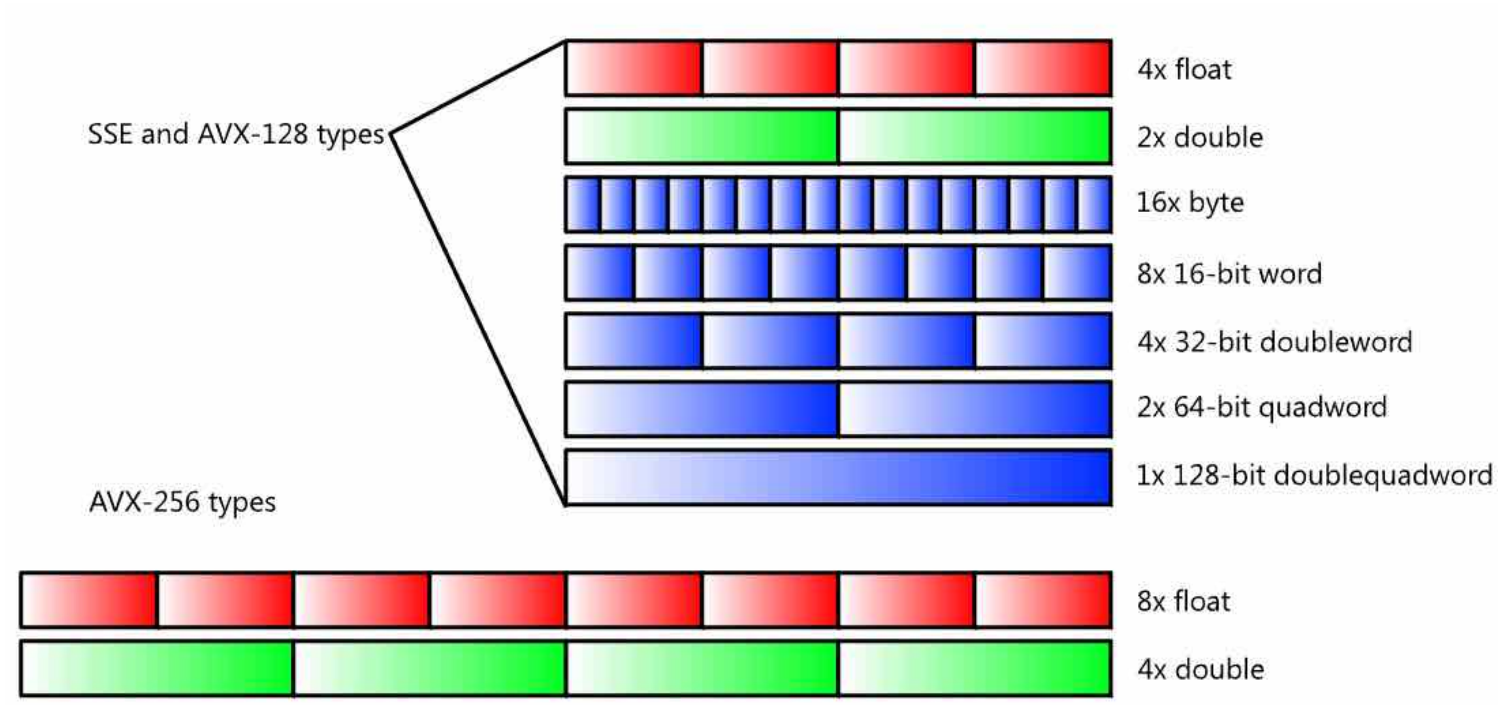
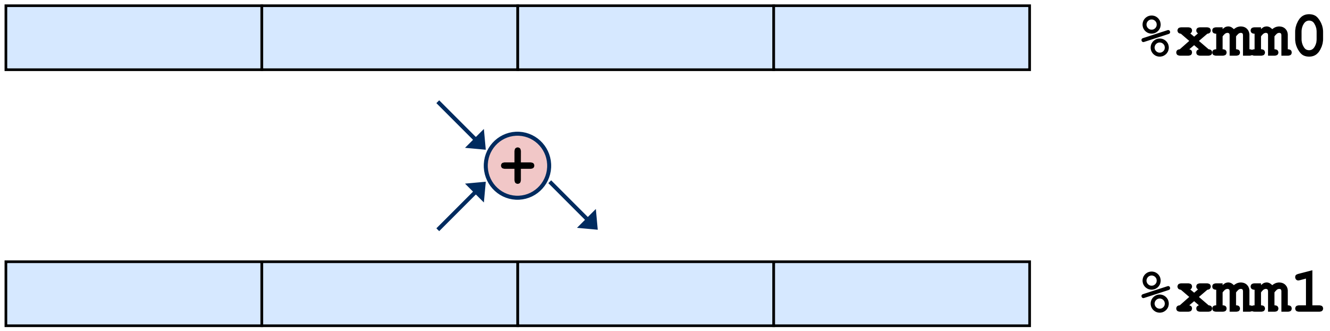


Figure 2. *Intel® AVX and Intel® SSE data types*

SSE3 Instructions: Examples

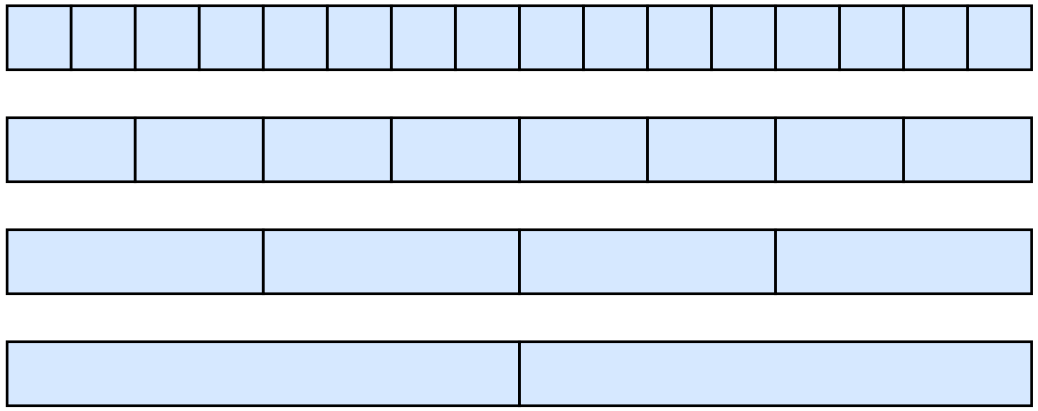
- Single precision *4-way vector add:* `addps %xmm0 %xmm1`



SSE Intrinsics (Focus Floating Point)

- **Data types**

```
__m128  f;   // = {float f0, f1, f2, f3}
__m128d d;   // = {double d0, d1}
__m128i i;   // 16 8-bit, 8 16-bit, 4 32-bit, or 2 64-bit ints
```



ints

ints

ints or floats

ints or doubles

AVX Intrinsics

- Data types

```
__m256    8 x float32
__m256d  4 x float64
```

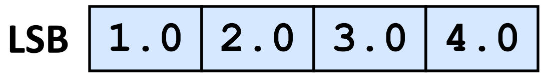
- No integer support!

SSE Intrinsics (Focus Floating Point)

- **Instructions**
  - Naming convention: `_mm_<intrin_op>_<suffix>`
  - Example:

```
// a is 16-byte aligned
float a[4] = {1.0, 2.0, 3.0, 4.0};
__m128 t = _mm_load_ps(a);
```

p: packed
s: single



  - **Same result as**

```
__m128 t = _mm_set_ps(4.0, 3.0, 2.0, 1.0)
```

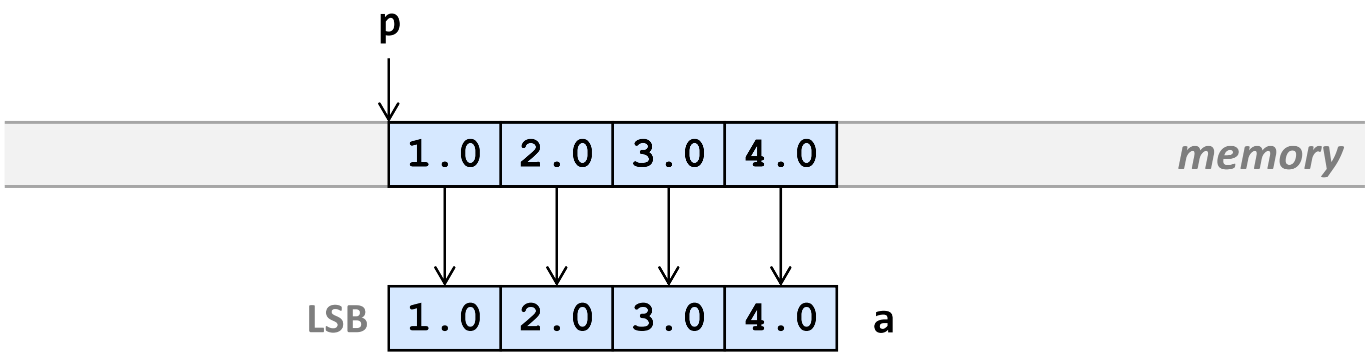
SSE Intrinsics

- **Native instructions (one-to-one with assembly)**
  `_mm_load_ps()`
  `_mm_add_ps()`
  `_mm_mul_ps()`
  …
- **Multi instructions (map to several assembly instructions)**
  `_mm_set_ps()`
  `_mm_set1_ps()`
  …
- **Macros and helpers**
  `_MM_TRANSPOSE4_PS()`
  `_MM_SHUFFLE()`
  …

What Are the Main Issues?

- Alignment is important (128 bit = 16 byte)
- You need to code explicit loads and stores
- **Don't mix SSE (128bit) with AVX (256bit)**

Loads and Stores



```
a = _mm_load_ps(p);  // p 16-byte aligned
```

```
a = _mm_loadu_ps(p); // p not aligned
```

avoid (expensive)

Arithmetic

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_add_ss	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

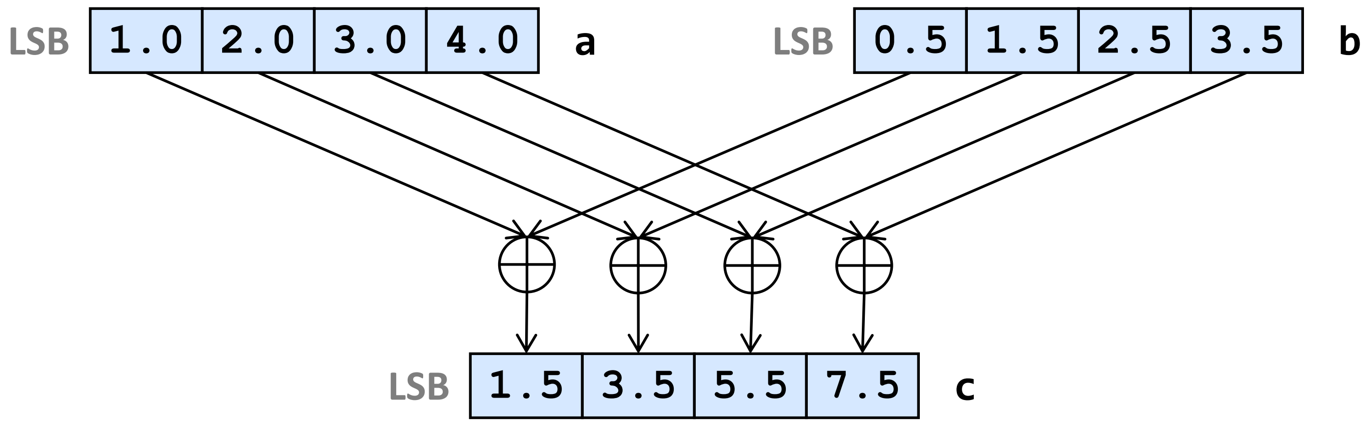
SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_addsub_ps	Subtract and add	ADDSUBPS
_mm_hadd_ps	Add	HADDPS
_mm_hsub_ps	Subtracts	HSUBPS

SSE4

Intrinsic	Operation	Corresponding SSE4 Instruction
_mm_dp_ps	Single precision dot product	DPPS

Arithmetic



```
c = _mm_add_ps(a, b);
```

analogous:

```
c = _mm_sub_ps(a, b);
```

```
c = _mm_mul_ps(a, b);
```

Example: Better Solution

```c
void addindex(float *x, int n) {
  for (int i = 0; i < n; i++)
    x[i] = x[i] + i;
}
```

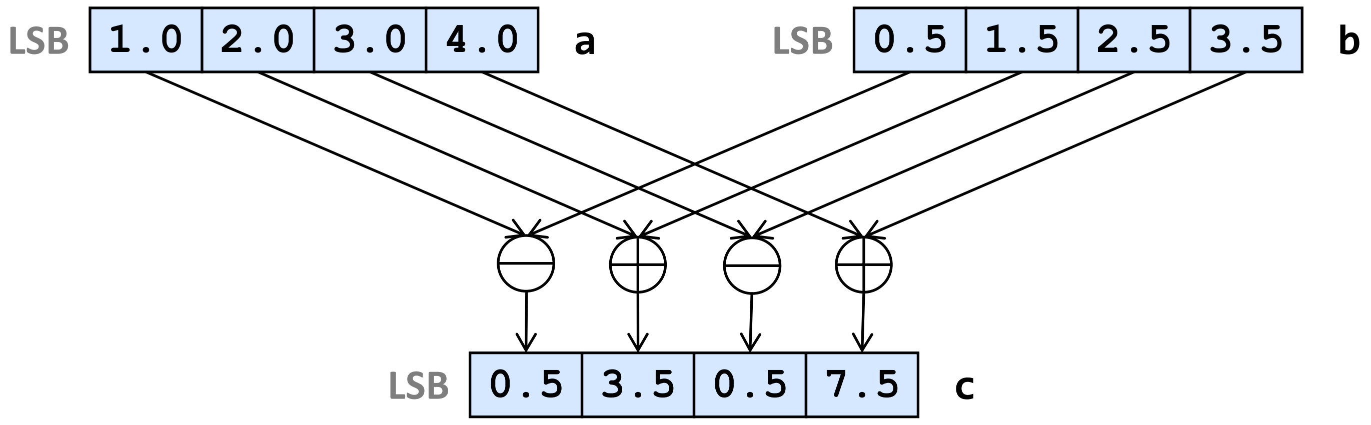
```c
#include <ia32intrin.h>

// n a multiple of 4, x is 16-byte aligned
void addindex_vec(float *x, int n) {
  __m128 index, incr, x_vec;

  index = _mm_set_ps(0, 1, 2, 3);
  incr  = _mm_set1_ps(4);
  for (int i = 0; i < n/4; i++) {
    x_vec = _mm_load_ps(x+i*4);         // load 4 floats
    x_vec = _mm_add_ps(x_vec, index);   // add index
    _mm_store_ps(x+i*4, x_vec);         // store back
    index = _mm_add_ps(index, incr);    // increment index
  }
}
```

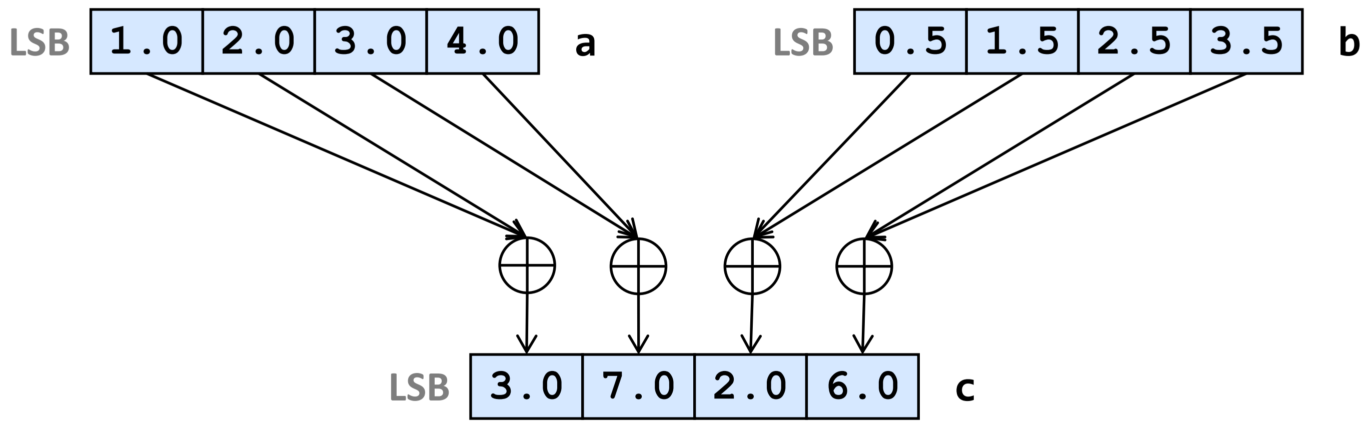
Note how using intrinsics implicitly forces scalar replacement!

Arithmetic



```
c = _mm_addsub_ps(a, b);
```

Arithmetic



```
c = _mm_hadd_ps(a, b);
```

analogous:

```
c = _mm_hsub_ps(a, b);
```

Shuffles

SSE

Intrinsic Name	Operation	Corresponding SSE Instruction
_mm_shuffle_ps	Shuffle	SHUFPS
_mm_unpackhi_ps	Unpack High	UNPCKHPS
_mm_unpacklo_ps	Unpack Low	UNPCKLPS
_mm_move_ss	Set low word, pass in three high values	MOVSS
_mm_movehl_ps	Move High to Low	MOVHLPS
_mm_movelh_ps	Move Low to High	MOVLHPS
_mm_movemask_ps	Create four-bit mask	MOVMSKPS

SSE3

Intrinsic Name	Operation	Corresponding SSE3 Instruction
_mm_movehdup_ps	Duplicates	MOVSHDUP
_mm_moveldup_ps	Duplicates	MOVSLDUP

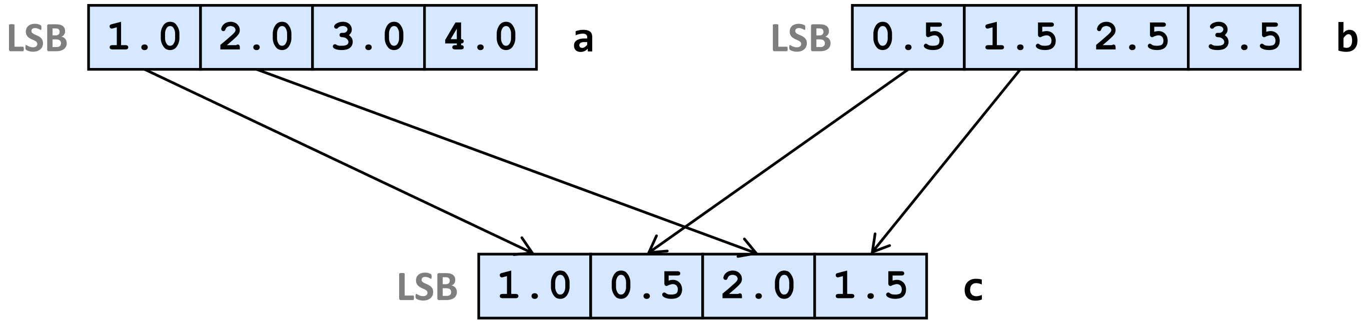
SSSE3

Intrinsic Name	Operation	Corresponding SSSE3 Instruction
_mm_shuffle_epi8	Shuffle	PSHUFB
_mm_alignr_epi8	Shift	PALIGNR

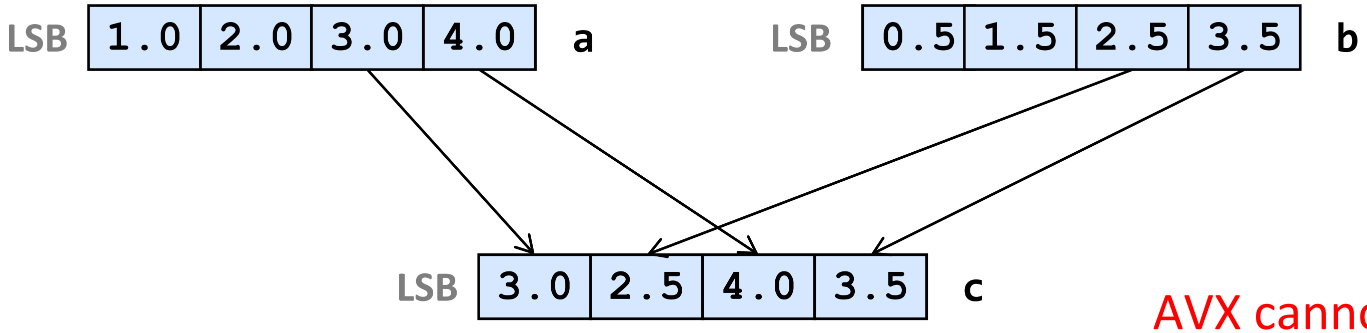
SSE4

Intrinsic Syntax	Operation	Corresponding SSE4 Instruction
__m128 _mm_blend_ps(__m128 v1, __m128 v2, const int mask)	Selects float single precision data from 2 sources using constant mask	BLENDPS
__m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3)	Selects float single precision data from 2 sources using variable mask	BLENDVPS
__m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx)	Insert single precision float into packed single precision array element selected by index.	INSERTPS
int _mm_extract_ps(__m128 src, const int ndx)	Extract single precision float from packed single precision array selected by index.	EXTRACTPS

Shuffles



```
c = _mm_unpacklo_ps(a, b);
```



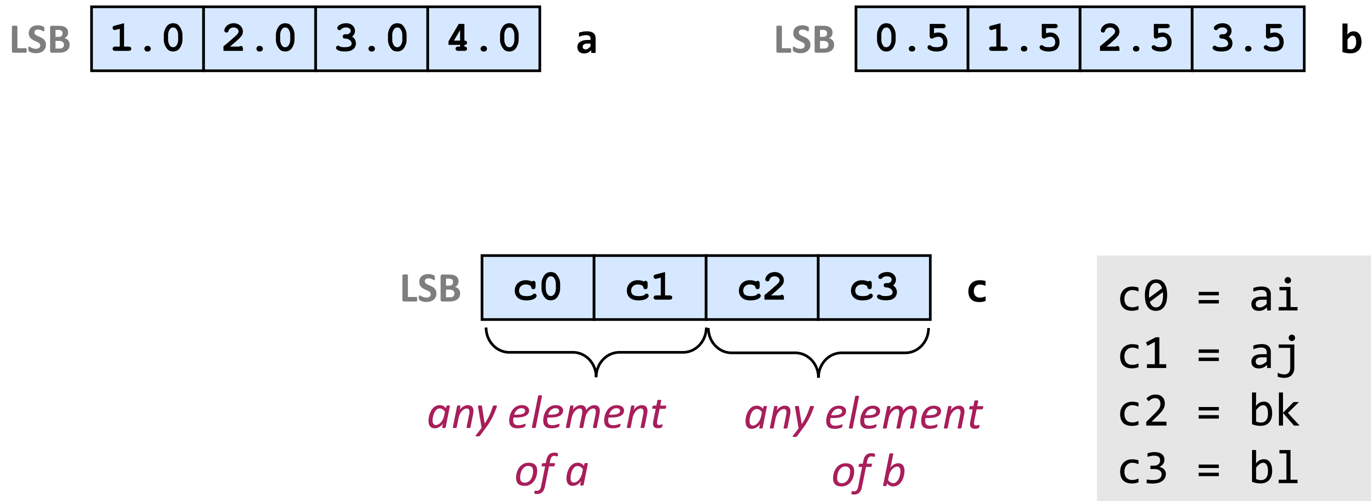
AVX cannot unpack between hi and low 128 bits

```
c = _mm_unpackhi_ps(a, b);
```

Shuffles

```
c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(l, k, j, i));
```

helper macro to create mask



```
c0 = ai
c1 = aj
c2 = bk
c3 = bl
i,j,k,l in {0,1,2,3}
```

AVX cannot shuffle between hi and low 128 bits

SIMD Essentials

- Include

```
#include <immintrin.h>
```

- Compile with

```
-msse2 -msse3 -mavx -mavx2
```

- Allocate memory aligned 16/32 bytes

```
int posix_memalign(void **memptr, size_t align, size_t size);

void* _mm_malloc (size_t size, size_t align );
void _mm_free (void *p)
```

DIFX: 32bit floating point vector add

```
int avxAdd_f32(f32 *src1, f32 *src2, f32 *dest, int length) {
  int i;
  float* ptrA = (float*)src1;
  float* ptrB = (float*)src2;
  float* ptrD = (float*)dest;

  for (i=0; i<length; i+= 8) {
    __m256 a      = _mm256_load_ps(ptrA);
    __m256 b      = _mm256_load_ps(ptrB);
    __m256 d      = _mm256_add_ps(a, b);
    _mm256_store_ps(ptrD, d);
    ptrA+=8;
    ptrB+=8;
    ptrD+=8;
  }
  return(0);
}
```

DIFX: Statistics

```
vecStatus avxMeanStdDev_f32(f32 *src, int length, f32 *mean,
f32 *StdDev) {
  int i;
  float sum8[8], s, ss;
  float *ptrA = src;

  __m256 sum = _mm256_set1_ps(0.0);
  __m256 sumsqr = _mm256_set1_ps(0.0);
  __m256 a;
  for (i=0; i<length; i+= 8) {
    a      = _mm256_load_ps(ptrA);
    sum    = _mm256_add_ps(sum, a);
    a      = _mm256_mul_ps(a,a);
    sumsqr = _mm256_add_ps(sumsqr, a);
    ptrA+=8;
  }
```

DIFX: Statistics (cont)

```
  // sum now contains the final 8 values which need to be averaged
  sum = _mm256_hadd_ps(sum,sum);
  __m256 sumshuffle = _mm256_permute2f128_ps(sum, sum, 0x1);
  sum = _mm256_add_ps(sum,sumshuffle);
  _mm256_store_ps(sum8, sum);
  s = sum8[0]+sum8[1];

  sumsqr = _mm256_hadd_ps(sumsqr,sumsqr);
  // Shuffle upper and lower 128bits
  sumshuffle = _mm256_permute2f128_ps(sumsqr, sumsqr, 0x1);
sumsqr = _mm256_add_ps(sumsqr,sumshuffle);
  _mm256_store_ps(sum8, sumsqr);
  ss = sum8[0]+sum8[1];
  *mean = s/length;
  *StdDev = sqrt((ss-(s*s/length))/(length-1));

  return(0);
}
```

DIFX: Multiplex

```
int avxRealToCplx_32f(f32 *src1, f32 *src2, cf32 *dest, int length) {
  int i;
  float* ptrA = (float*)src1;
  float* ptrB = (float*)src2;
  float* ptrD = (float*)dest;

  for (i=0; i<length; i+= 8) {
    __m256 a      = _mm256_load_ps(ptrA);
    __m256 b      = _mm256_load_ps(ptrB);
    __m256 i1     = _mm256_unpacklo_ps(a, b); // 3rd and 4th entries wrong order
    __m256 i2     = _mm256_unpackhi_ps (a, b); // 1st and second entries, wrong order
    __m256 d      = _mm256_permute2f128_ps(i1, i2, 0x20); // Shuffle upper and lower 128bits
    _mm256_store_ps(ptrD, d);
    ptrD+=8;
           d      = _mm256_permute2f128_ps(i1, i2, 0x31); // Shuffle upper and lower 128bits
    _mm256_store_ps(ptrD, d);
    ptrA+=8;
    ptrB+=8;
    ptrD+=8;
  }
  return(0);
}
```

DIFX: Complex AddProduct

```
int  avxAddProduct_cf32(cf32 *src1, cf32 *src2, cf32 *dest, int length) {
  int i;
  float* ptrA = (float*)src1;
  float* ptrB = (float*)src2;
  float* ptrD = (float*)dest;
  for (i=0; i<length; i+= 4) {
    __m256 a      = _mm256_load_ps(ptrA);          // (a.re, a.im) x4
    __m256 b      = _mm256_load_ps(ptrB);          // (b.re, b.im) x4
    __m256 c      = _mm256_load_ps(ptrD);
    __m256 b_flip = _mm256_shuffle_ps(b,b,0xB1);   // (b.im, b.re) x4
    __m256 a_im   = _mm256_shuffle_ps(a,a,0xF5);   // (a.im, a.im) x4
    __m256 a_re   = _mm256_shuffle_ps(a,a,0xA0);   // (a.re, a.re) x4
    __m256 aib    = _mm256_mul_ps(a_im, b_flip);   // (a.im*b.im, a.im*b.re) x4
    __m256 arb    = _mm256_mul_ps(a_re, b);        // (a.re*b.re, a.re*b.im) x4
    __m256 prod   = _mm256_addsub_ps(arb, aib);    // Actual product
    __m256 D      = _mm256_add_ps(prod, c);        // Accumulate
    _mm256_store_ps(ptrD, D);
    ptrA+=8;
    ptrB+=8;
    ptrD+=8;
  }
  return(0);
}
```

Benchmarks (2.3 GHz Core i7)

	Generic	IPP	SSE	AVX
Float Add	1.9 sec	1.9 sec	1.8 sec	1.9 sec
Complex AddProduct	3.2 sec	1.8 sec	1.8 sec	1.8 sec
Float MeanStdDev	2.7 sec	1.0 sec	0.7 sec	0.7 sec
Float->Complex	1.6 sec	1.1 sec	1.1 sec	1.1 sec

Suggested DIFX Route

- Can replace all IPP vector code with hand coded intrinsics with a couple of days work
  - Can also have hybrid generic/SIMD
- Suggest 3 or 4 flavors
  - Generic
  - IPP
  - SSE3 (SSE4?)
  - AVX?
- No speed improvement for simpleSIMD replacement
- Consider if combined functions make sense
  - Possible significant speed improvements

Resources

- Intel intrinsics reference
  https://software.intel.com/en-us/node/513410
- Intel Developers Manual
  http://www.intel.com.au/content/www/au/en/processors/architectures-software-developer-manuals.html
- Memory Management
  https://software.intel.com/en-us/articles/memory-management-for-optimal-performance-on-intel-xeon-phi-coprocessor-alignment-and

Thank you

Astronomy and Space Science
Chris Phillips
LBA Lead Scientist

t +61 2 9372 4608
e Chris.Phillips@csiro.au
w www.atnf.csiro.au