# SSE/SSE2 Toolbox Solutions for Real-Life SIMD Problems

**Alex.Klimovitski@intel.com**
**Tools & Technologies Europe**
**Intel Corporation**

**Game Developer Conference 2001**

intel®

# Why SIMD? Why SSE?

- SIMD is the primary performance feature on most platforms
- SSE is a powerful and versatile implementation of SIMD
- offered on mass platforms, all Pentium® III and Pentium 4 PCs, Xbox, recent Athlon* PCs

intel®

# Agenda

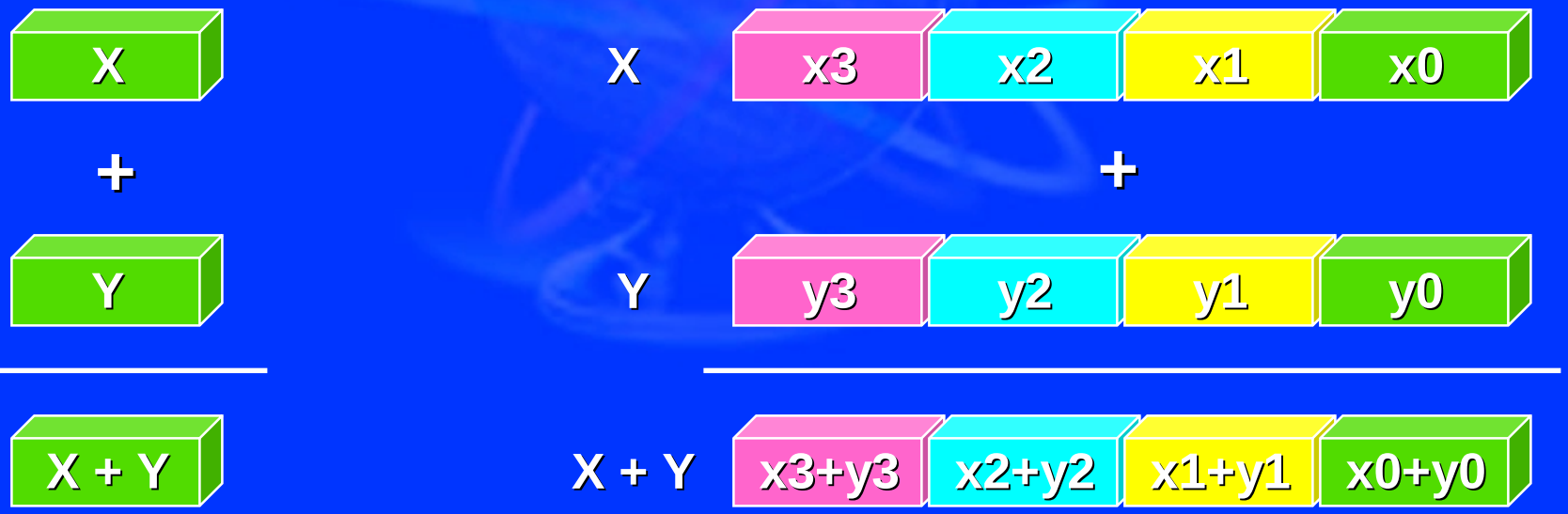
- Exploiting Parallelism
- Data Restructuring
- Data Compression
- Conditional Code with SIMD
- Summary
- Bonus Foils

# Agenda

- Exploiting Parallelism
- Data Restructuring
- Data Compression
- Conditional Code with SIMD
- Summary
- Bonus Foils

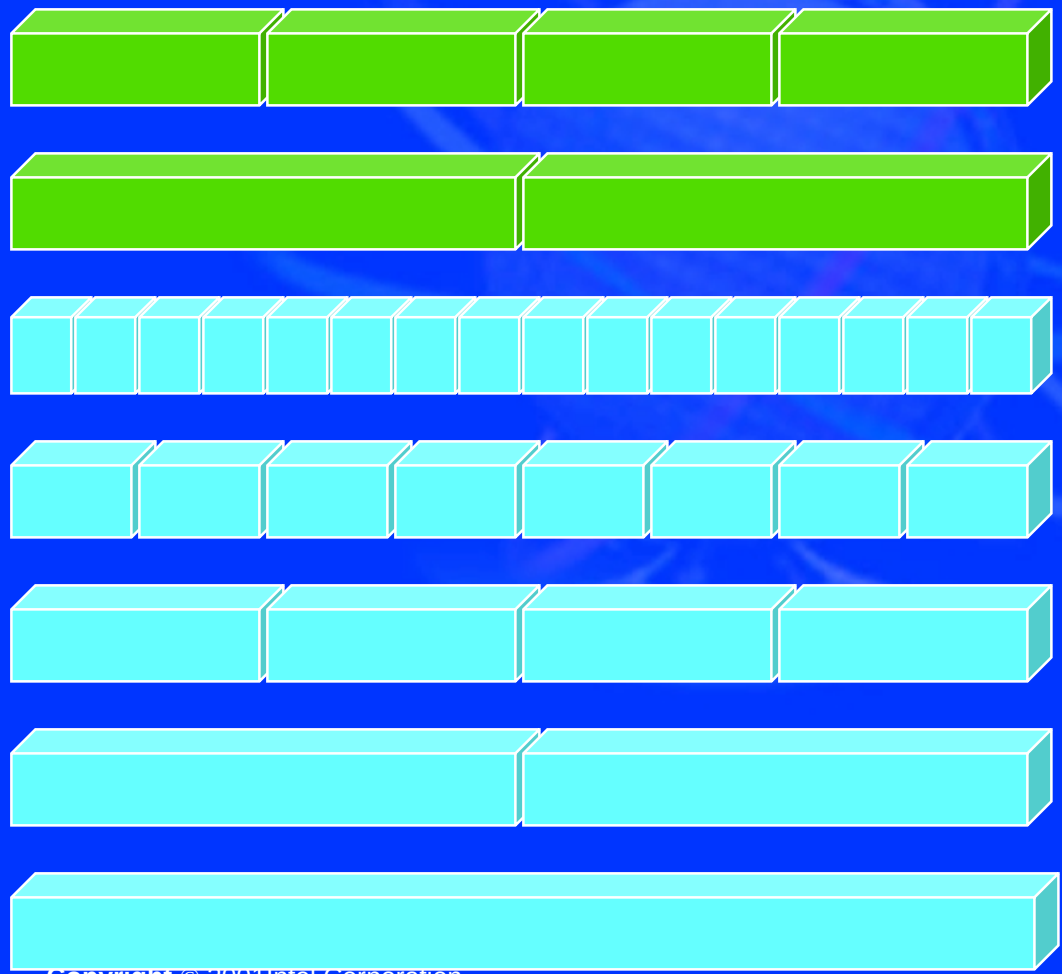
# Introducing SIMD: Single Instruction, Multiple Data

- **Scalar processing**
  - traditional mode
  - one operation produces one result
- **SIMD processing**
  - with SSE / SSE2
  - one operation produces multiple results

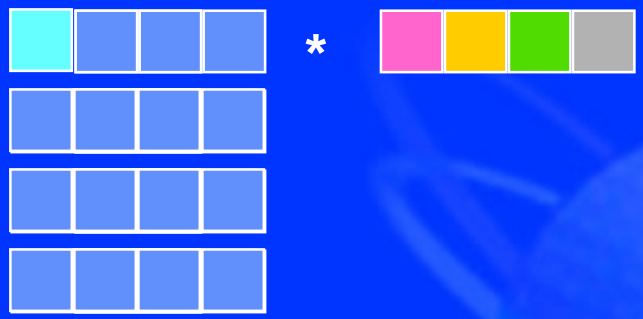


# SSE / SSE2 SIMD Data Types

- Anything that fits into 16 byte!



# Matrix by Vector Example

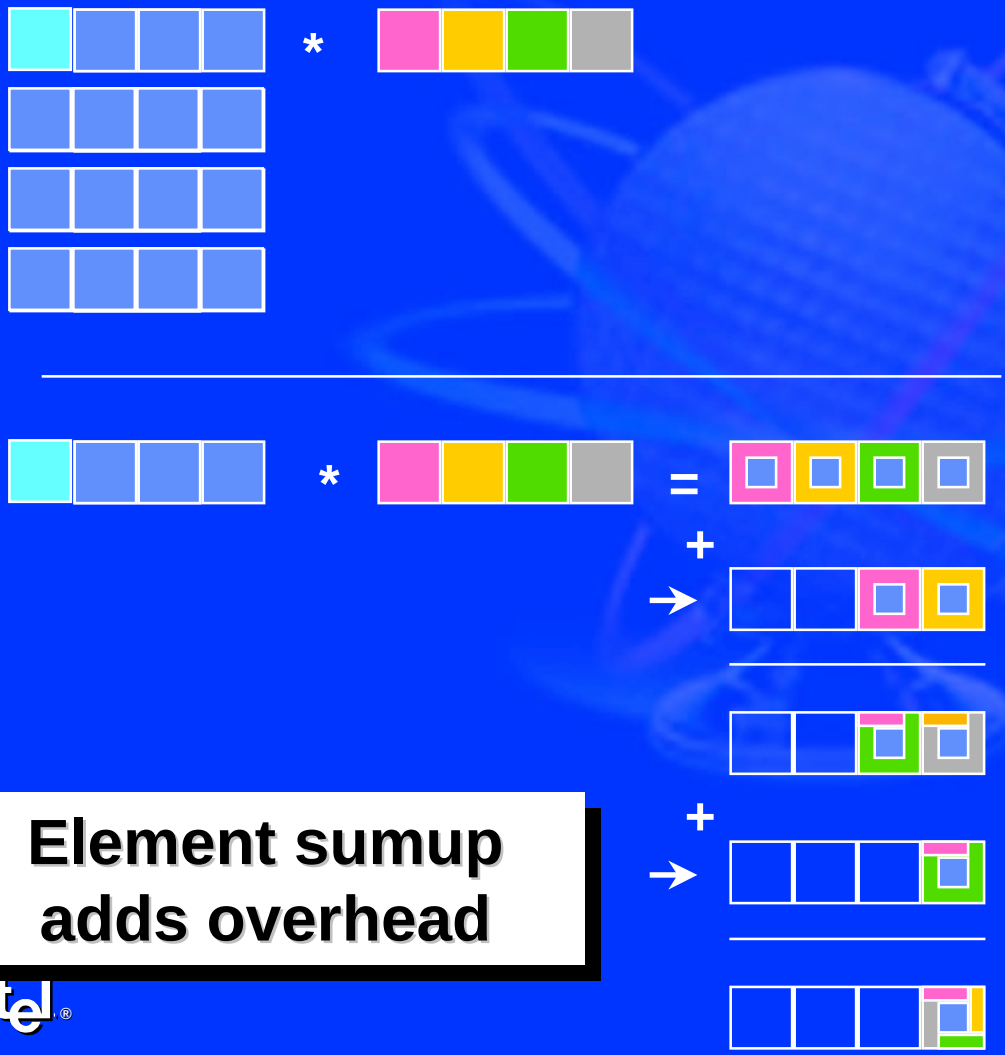


For X, Y, Z, W:

# M by V Code

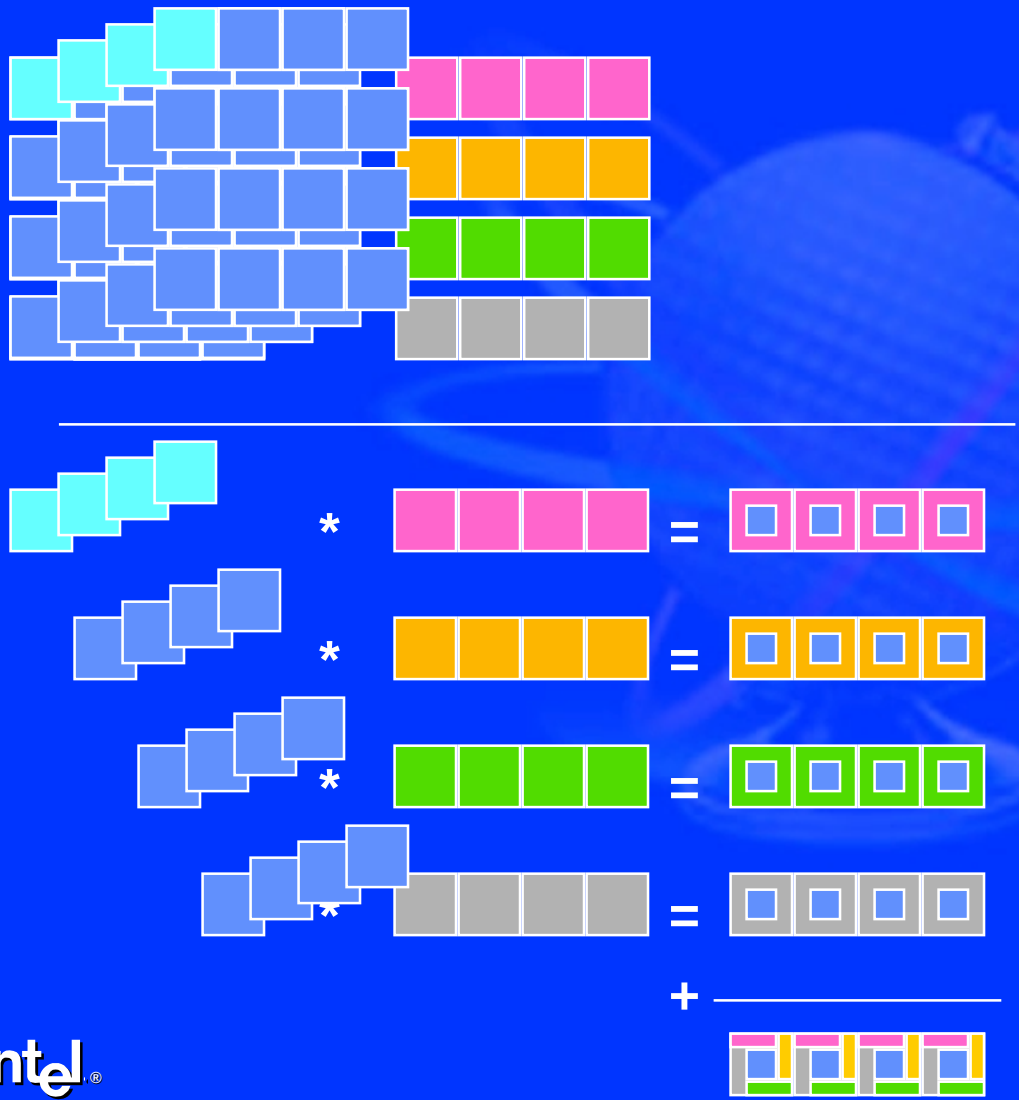
```
static float m[4][4];

for (int i = 0; i < ARRAY_COUNT; i++) {
    float x = xi[i];
    float y = yi[i];
    float z = zi[i];
    float w = wi[i];
    xo[i] = x * m[0][0] + y * m[0][1] + z * m[0][2] +
        w * m[0][3];
    yo[i] = x * m[1][0] + y * m[1][1] + z * m[1][2] +
        w * m[1][3];
    zo[i] = x * m[2][0] + y * m[2][1] + z * m[2][2] +
        w * m[2][3];
    wo[i] = x * m[3][0] + y * m[3][1] + z * m[3][2] +
        w * m[3][3];
}
```

# M by V with SSE, 1st Try

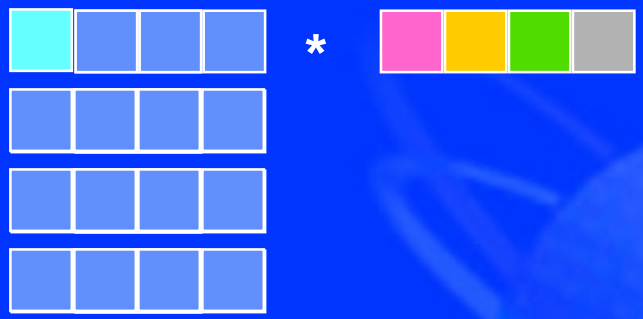


**Element sumup adds overhead**

# M by V with SSE, 2nd Try



# Same Operation – Just Four at A Time!





For X, Y, Z, W:

# M by V Code with SSE

```
static F32vec4 q[4][4];

for (int i = 0; i < ARRAY_COUNT; i += 4) {
    F32vec4 x = (F32vec4&)xi[i];
    F32vec4 y = (F32vec4&)yi[i];
    F32vec4 z = (F32vec4&)zi[i];
    F32vec4 w = (F32vec4&)wi[i];
    (F32vec4&)xo[i] = x * q[0][0] + y * q[0][1] +
       z * q[0][2] + w * q[0][3];
    (F32vec4&)yo[i] = x * q[1][0] + y * q[1][1] +
       z * q[1][2] + w * q[1][3];
    (F32vec4&)zo[i] = x * q[2][0] + y * q[2][1] +
       z * q[2][2] + w * q[2][3];
    (F32vec4&)wo[i] = x * q[3][0] + y * q[3][1] +
       z * q[3][2] + w * q[3][3];
}
```

# Same Code as Scalar – Just Four at A Time!

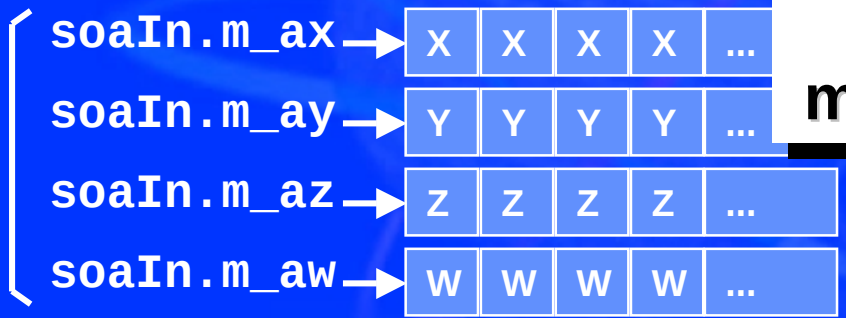
```
static float m[4][4];

for (int i = 0; i < ARRAY_COUNT; i++) {
    float x = xi[i];
    float y = yi[i];
    float z = zi[i];
    float w = wi[i];
    xo[i] = x * m[0][0] + y * m[0][1] + z * m[0][2] +
        w * m[0][3];
    yo[i] = x * m[1][0] + y * m[1][1] + z * m[1][2] +
        w * m[1][3];
    zo[i] = x * m[2][0] + y * m[2][1] + z * m[2][2] +
        w * m[2][3];
    wo[i] = x * m[3][0] + y * m[3][1] + z * m[3][2] +
        w * m[3][3];
}
```

# Remember Memory

- AoS: Array of Structures

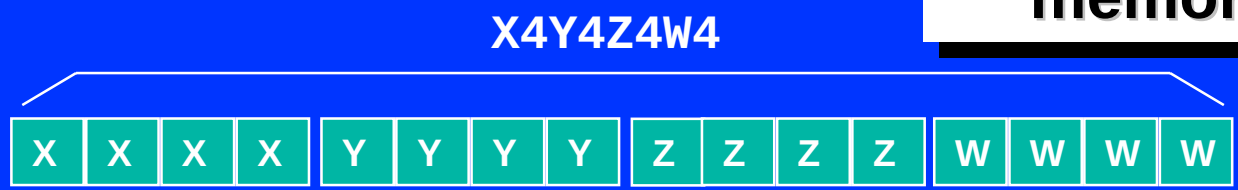

**AoS defeats SIMD**

- SoA: Structure of Arrays




**SoA provides for maximum parallelism!**

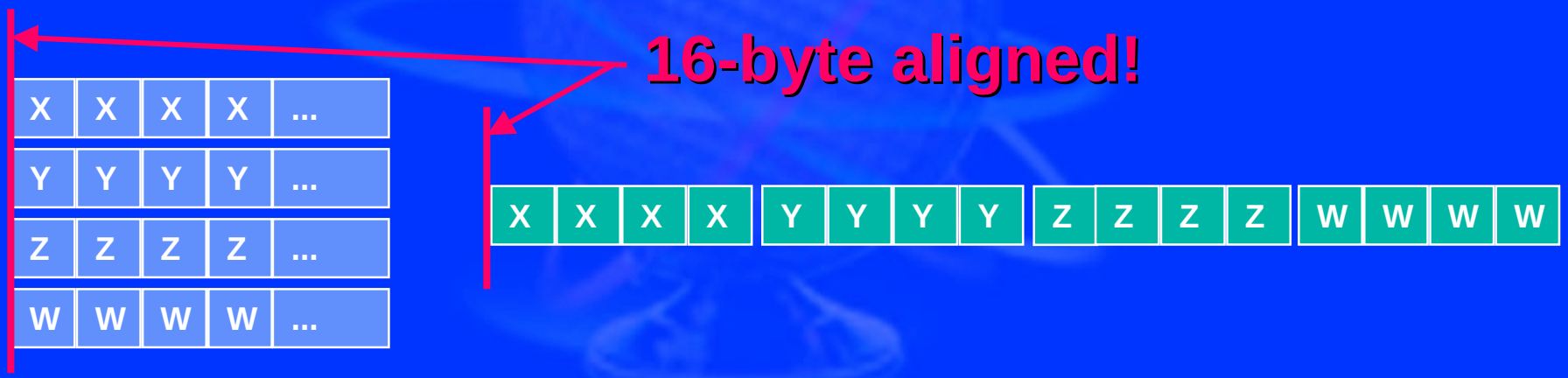
- Hybrid Structure




**Hybrid is also more memory-friendly!**

# Remember Alignment, too!

- SSE/SSE2 loads/store expect data aligned on 16-byte boundary; otherwise crash!
- There are unaligned load/store versions, but these are significantly slower




```
__declspec(align(16)) float a[N]; // static or auto
int* b = _mm_malloc(N * sizeof(int), 16); // dynamic
_mm_free(b);
F32vec4 c[N / 4]; // Vec Classes are always aligned
```

# M by V with Perspective Correction Code

```
for (int i = 0; i < ARRAY_COUNT; i++) {
    float x = xi[i];
    float y = yi[i];
    float z = zi[i];
    float w = wi[i];

    float wr = 1.0 / (x * m[3][0] + y * m[3][1] +
        z * m[3][2] + w * m[3][3]);

    xo[i] = wr * (x * m[0][0] + y * m[0][1] +
        z * m[0][2] + w * m[0][3]);
    yo[i] = wr * (x * m[1][0] + y * m[1][1] +
        z * m[1][2] + w * m[1][3]);
    zo[i] = wr * (x * m[2][0] + y * m[2][1] +
        z * m[2][2] + w * m[2][3]);
    wo[i] = wr;
}
```

# M by V with Perspective Correction SSE Code

```
for (int i = 0; i < ARRAY_COUNT; i += 4) {
    F32vec4 x = (F32vec4&)xi[i];
    F32vec4 y = (F32vec4&)yi[i];
    F32vec4 z = (F32vec4&)zi[i];
    F32vec4 w = (F32vec4&)wi[i];

    F32vec4 wr = rcp_nr(x * q[3][0] + y * q[3][1] +
        z * q[3][2] + w * q[3][3]);

    (F32vec4&)xo[i] = wr * (x * q[0][0] + y * q[0][1]
        + z * q[0][2] + w * q[0][3]);
    (F32vec4&)yo[i] = wr * (x * q[1][0] + y * q[1][1]
        + z * q[1][2] + w * q[1][3]);
    (F32vec4&)zo[i] = wr * (x * q[2][0] + y * q[2][1]
        + z * q[2][2] + w * q[2][3]);
    (F32vec4&)wo[i
}
```

**Easy per-component processing!**

# “SIMDizing” The Matrix

```
void FourFloats2F32vec4(F32vec4* v, const float* f)
{
 v[0]=_mm_load_ps(f);
 v[1]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(1,1,1,1));
 v[2]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(2,2,2,2));
 v[3]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(3,3,3,3));
 v[0]=_mm_shuffle_ps(v[0],v[0],_MM_SHUFFLE(0,0,0,0));
}

static _MM_ALIGN16 float m[4][4];
static F32vec4 q[4][4];

for (int i = 0; i < 4; i++)
 FourFloats2F32vec4(q[i], m[i]);
```

**Align scalar data, too!**

# Rules of Good Parallelism

- Maintain the original algorithm
- Process {four} data portions in parallel
- Keep only homogeneous components in one SIMD operand
- {Quadruple} loop-invariants outside the loop to create SIMD invariants
- Use SIMD-friendly structure, SoA or Hybrid

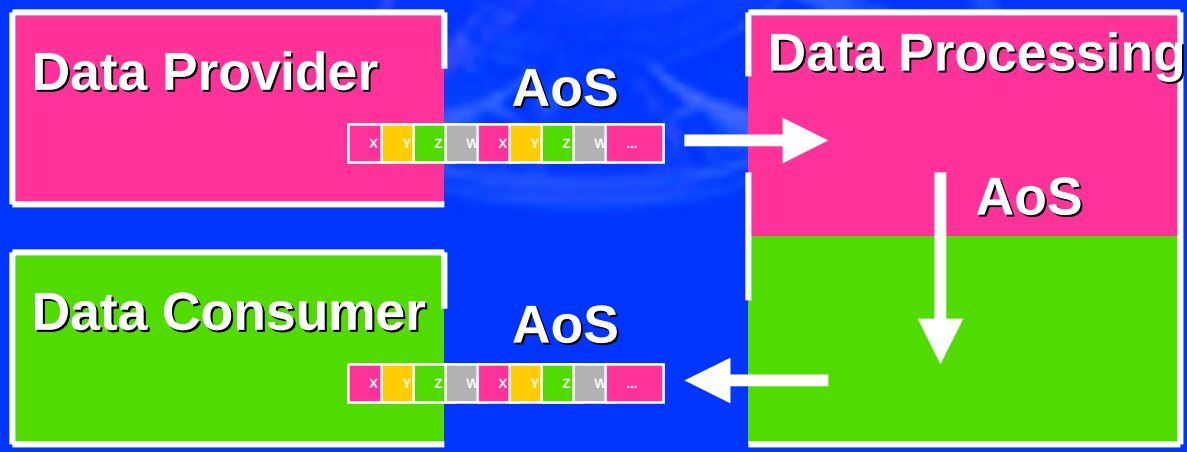
intel®

# Agenda

- ✓ Exploiting Parallelism
- ➢ Data Restructuring
- Data Compression
- Conditional Code with SIMD
- Summary
- Bonus Foils

# SIMD-Unfriendly Data Structures

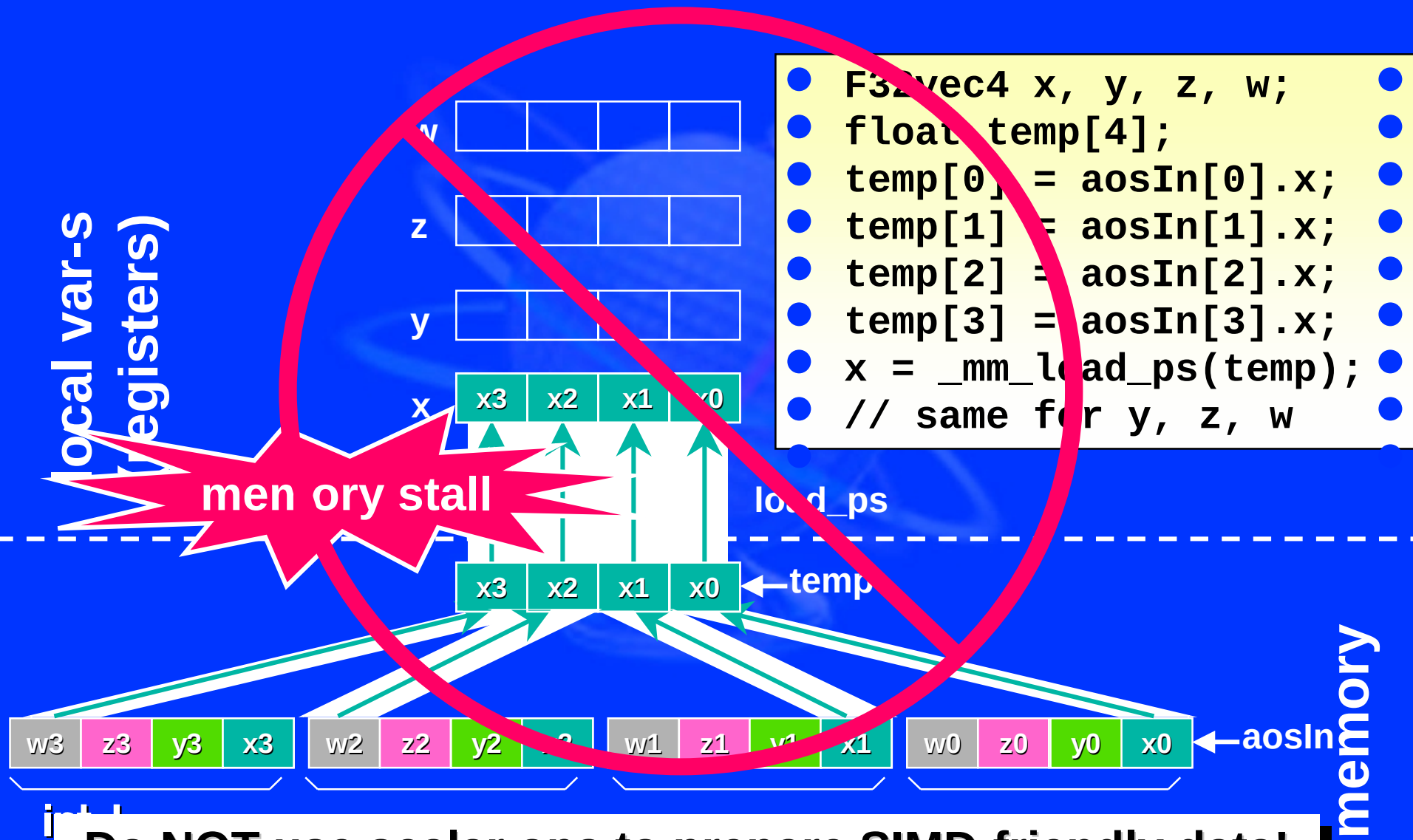
- The primary SIMD problem
- Results from:
  - Interface / API Constraints
  - Algorithm Logic
  - Legacy Code



# Taming SIMD-Unfriendly Data

- "Swizzle" (transform) data at run-time
- Pre-swizzle at design/load-time as much as possible
- Implement swizzle with SSE / SSE2

# Data Swizzling Wrong Way

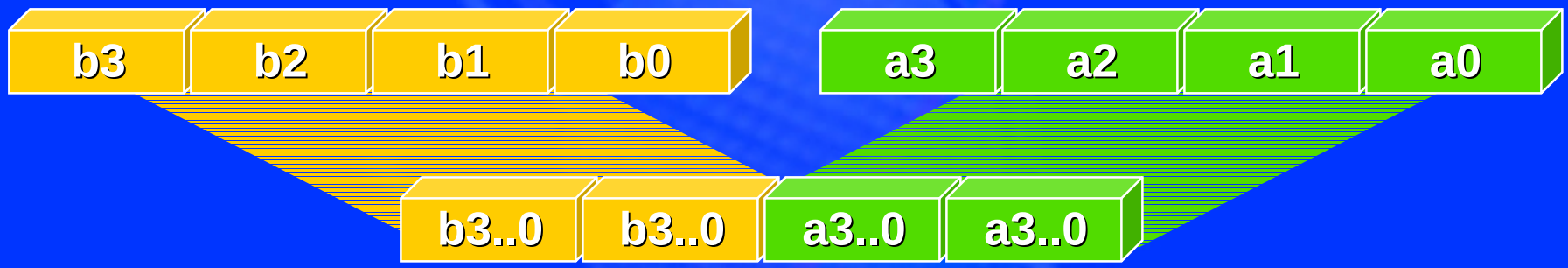



**Do NOT use scalar ops to prepare SIMD-friendly data!**

# Chief SIMD Swizzler: Shuffle

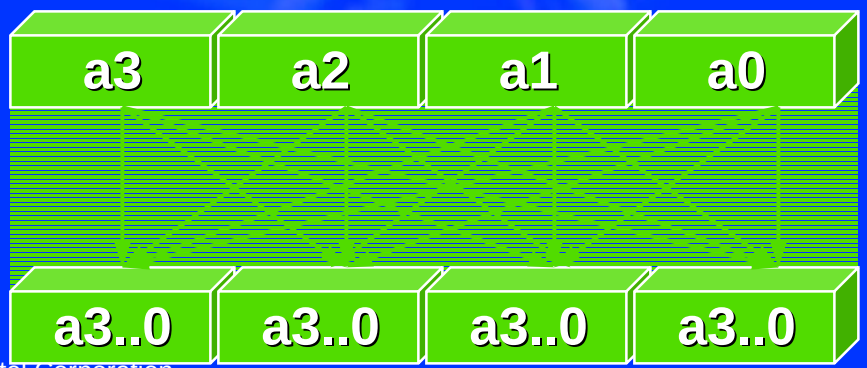
- First operand contributes two lower elements, second operand contributes two higher ones

```
_mm_shuffle_ps(a, b, _MM_SHUFFLE(3,1,2,0))
```

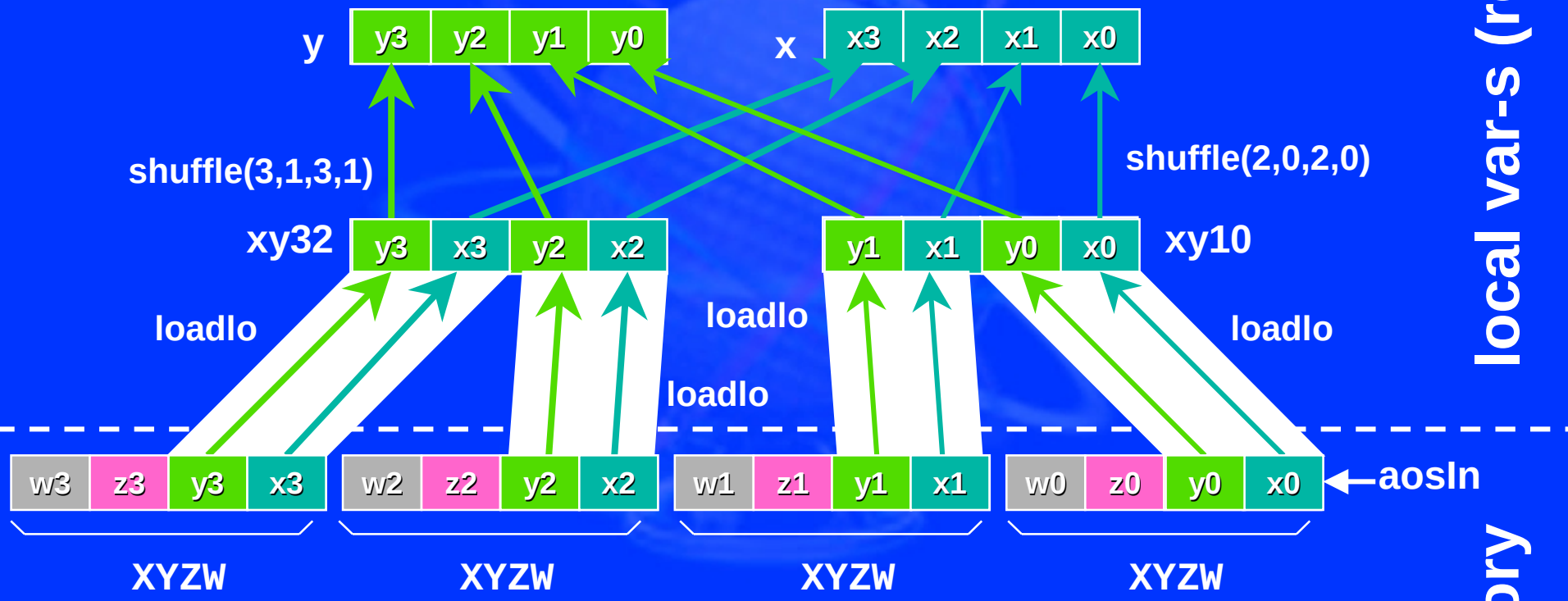



- Shuffle with itself: total swizzle

```
_mm_shuffle_ps(a, a, _MM_SHUFFLE(3,1,2,0))
```

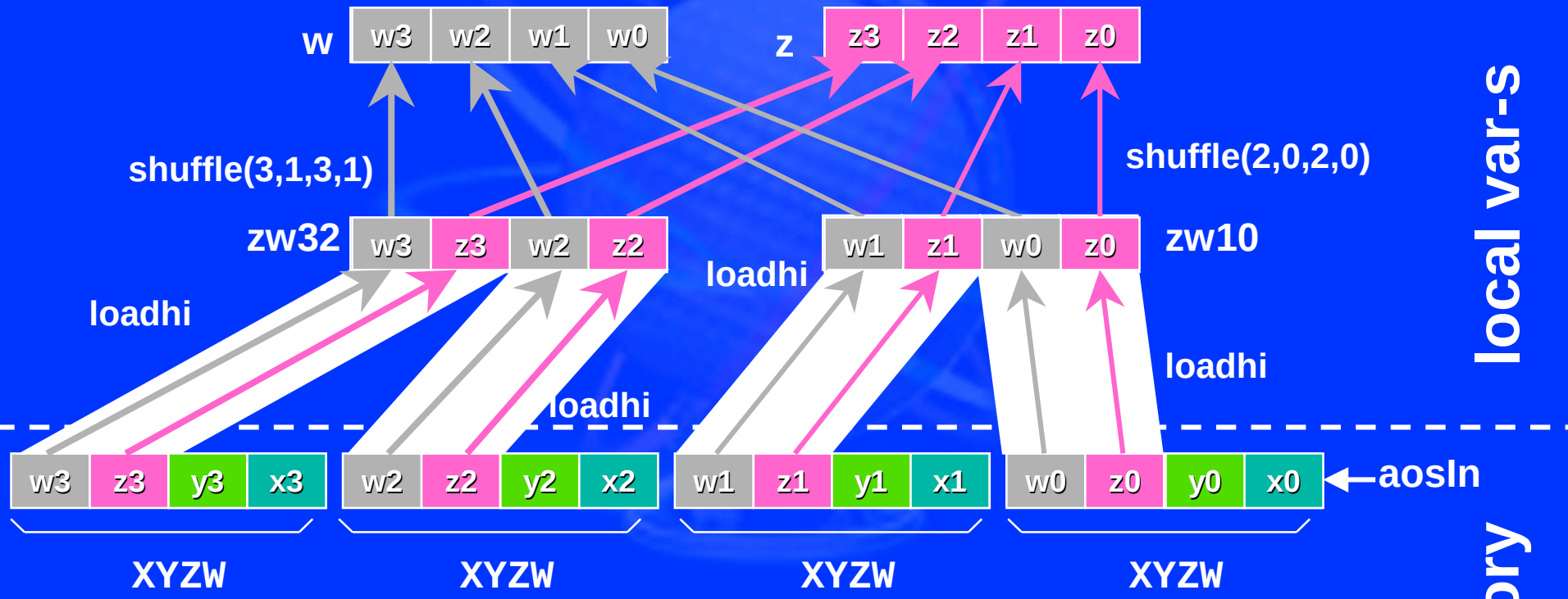


# Data Swizzling with SIMD: AoS to SoA




**Similar steps for z, w!**

# Data Swizzling with SIMD: AoS to SoA, Continued



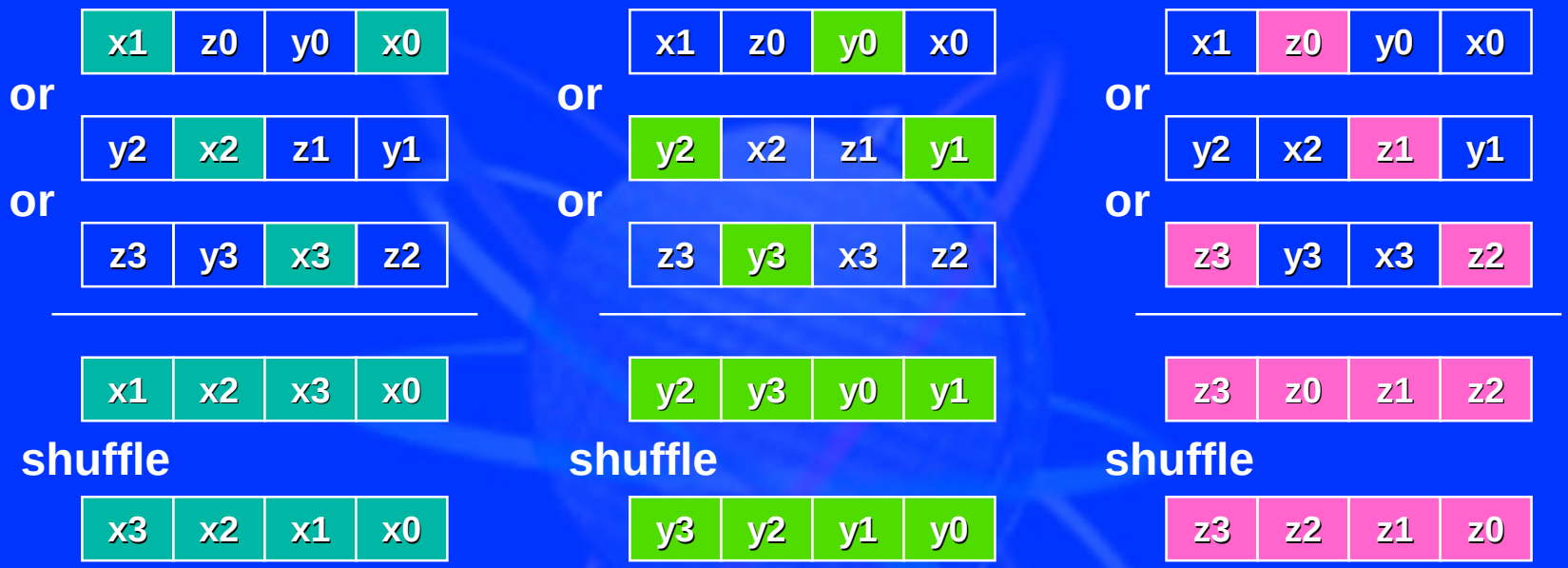

**To prepare SIMD-friendly data, use SIMD!**

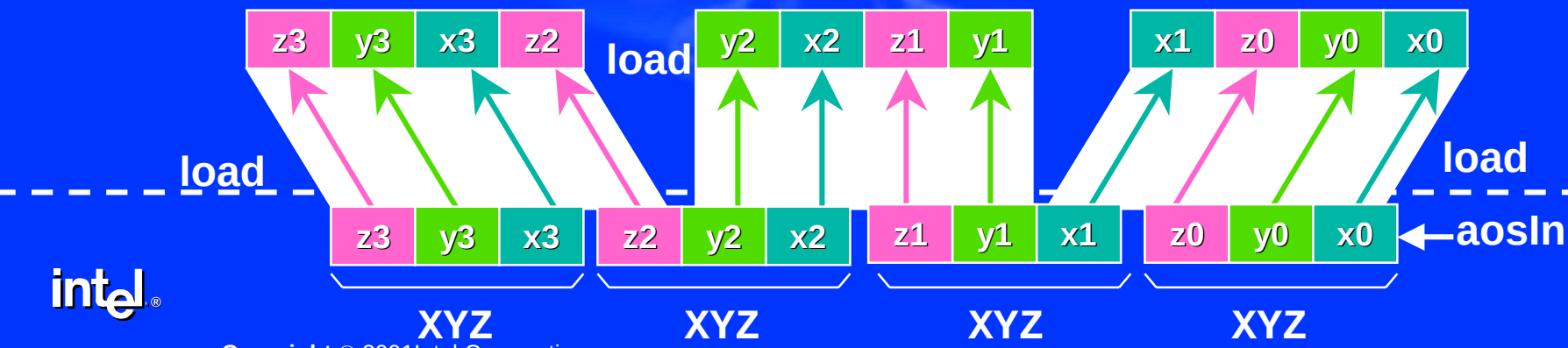
intel®

# AoS to SoA Code

```
void XYZWtoF32vec4(F32vec4& x, y, z, w, XYZW* aosIn)
{
 F32vec4 xy10, xy32, zw10, zw32;
 xy10 = zw10 = _mm_setzero_ps();
 xy10 = _mm_loadl_pi(xy10, (__m64*)&(aosIn[0]).x);
 zw10 = _mm_loadl_pi(zw10, (__m64*)&(aosIn[0]).z);
 xy10 = _mm_loadh_pi(xy10, (__m64*)&(aosIn[1]).x);
 zw10 = _mm_loadh_pi(zw10, (__m64*)&(aosIn[1]).z);
 xy32 = zw32 = _mm_setzero_ps();
 xy32 = _mm_loadl_pi(xy32, (__m64*)&(aosIn[2]).x);
 zw32 = _mm_loadl_pi(zw32, (__m64*)&(aosIn[2]).z);
 xy32 = _mm_loadh_pi(xy32, (__m64*)&(aosIn[3]).x);
 zw32 = _mm_loadh_pi(zw32, (__m64*)&(aosIn[3]).z);
 x = _mm_shuffle_ps(xy10, xy32, SHUFFLE(2,0,2,0));
 y = _mm_shuffle_ps(xy10, xy32, SHUFFLE(3,1,3,1));
 z = _mm_shuffle_ps(zw10, zw32, SHUFFLE(2,0,2,0));
 w = _mm_shuffle_ps(zw10, zw32, SHUFFLE(3,1,3,1));
}
```

# 3-Component AoS to SoA



# Defining SSE Bit Masks

- No valid `float` with bit pattern needed?
- Define aligned static array of four integers
- Load it at runtime as packed `floats`
- Implemented as a macro `CONST_INT32_PS`

```
#define CONST_INT32_PS(N, V3,V2,V1,V0) \
static const _MM_ALIGN16 int _##N[]= \
    {V0, V1, V2, V3};/*little endian!*/ \
const F32vec4 N = _mm_load_ps((float*)_##N);

// usage example, mask for elements 3 and 1:
CONST_INT32_PS(mask31,  ~0, 0, ~0, 0);
```

intel®

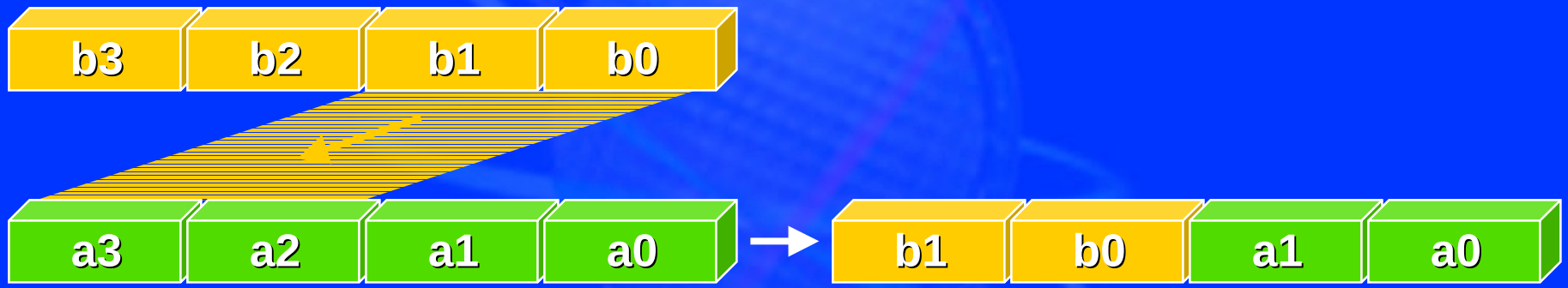
# Swizzling 3-Component AoS to SoA Code

```
void XYZtoF32vec4(F32vec4& x, y, z, XYZ* aosIn)
{
 F32vec4 a, b, c;
 CONST_INT32_PS(mask30,  ~0, 0, 0, ~0); // etc.

 a = _mm_load_ps((float*)aosIn);
 b = _mm_load_ps(((float*)aosIn) + 4);
 c = _mm_load_ps(((float*)aosIn) + 8);

 x = (a & mask30) | (b & mask2) | (c & mask1);
 y = (a & mask1) | (b & mask30) | (c & mask2);
 z = (a & mask2) | (b & mask1) | (c & mask30);

 x = _mm_shuffle_ps(x, x, _MM_SHUFFLE(1,2,3,0));
 y = _mm_shuffle_ps(y, y, _MM_SHUFFLE(2,3,0,1));
 z = _mm_shuffle_ps(z, z, _MM_SHUFFLE(3,0,1,2));
}
```

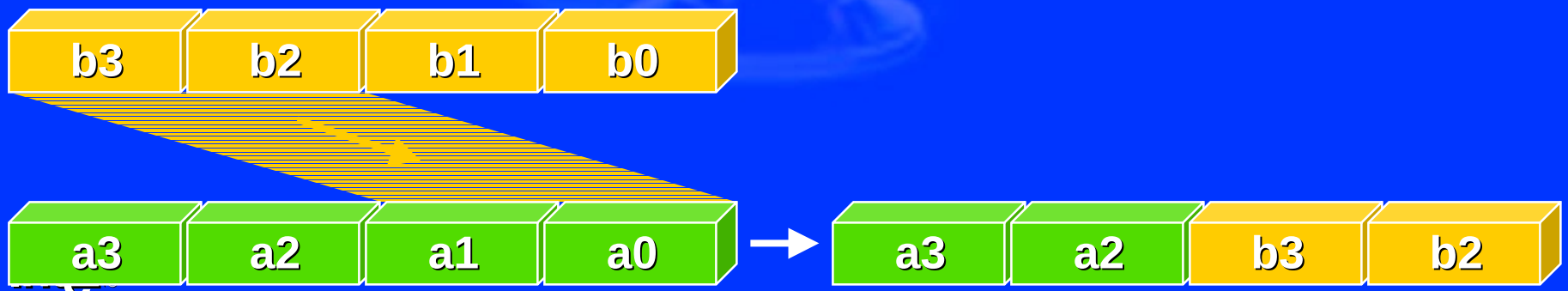
# Gatherers: Cross-Half-Moves

- Move lower (higher) half of the second operand to higher (lower) half of the first operand

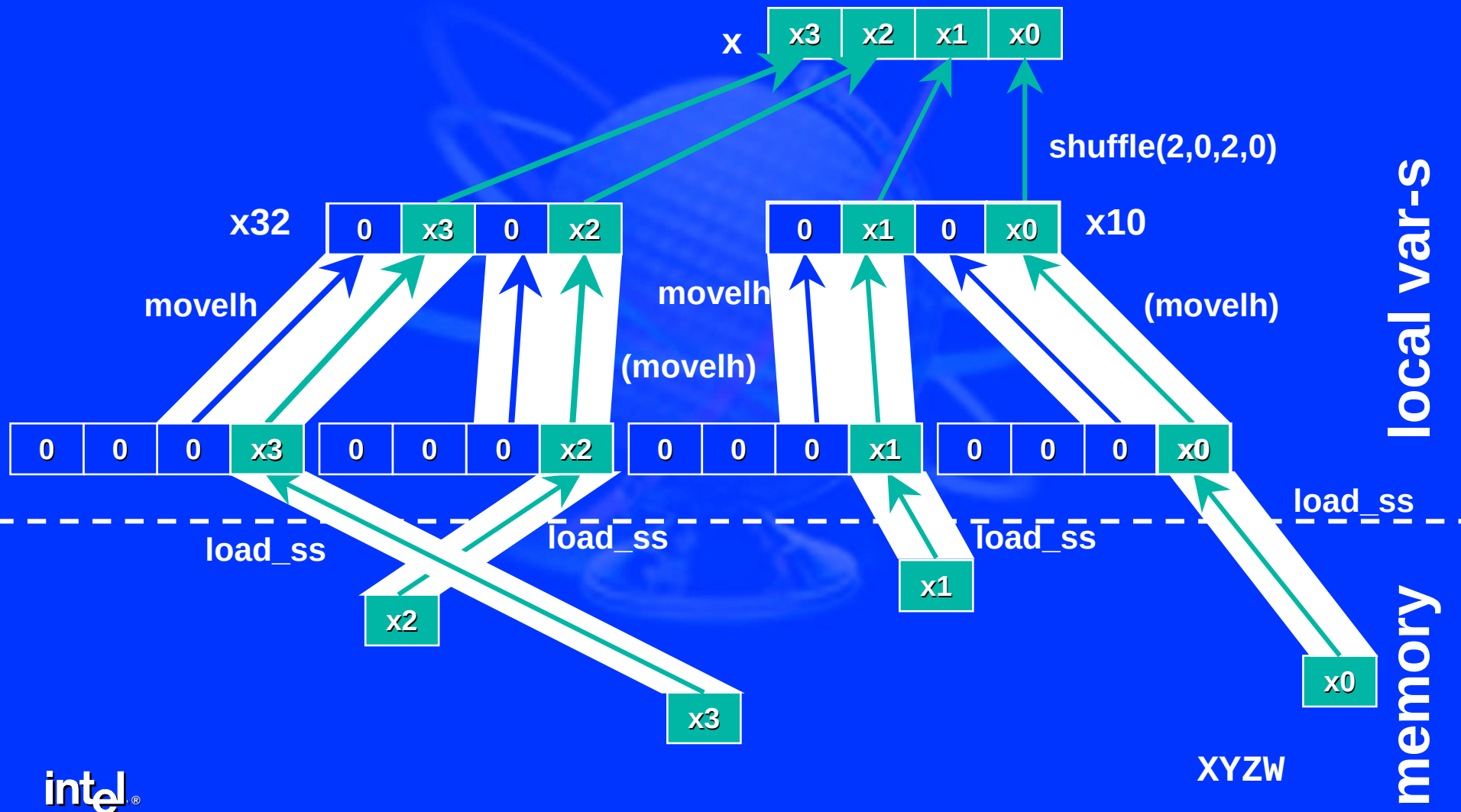
`_mm_movelh_ps(a, b)`




`_mm_movehl_ps(a, b)`



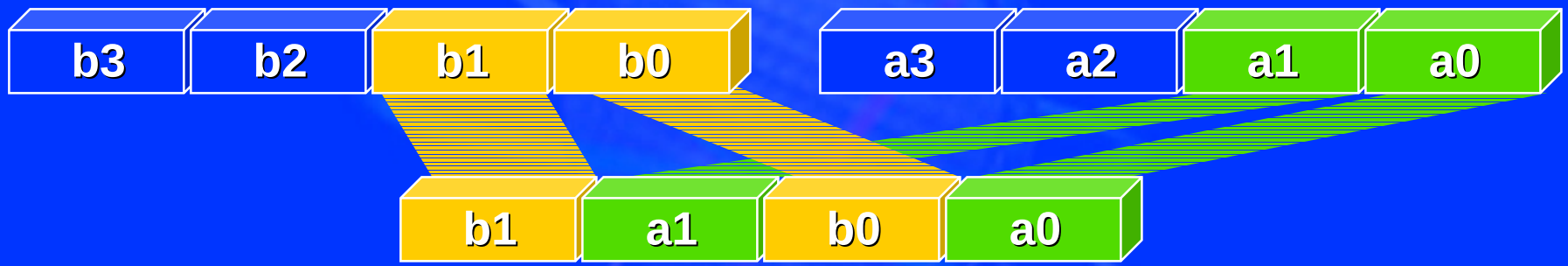
# Scatter-Gathering + Swizzling



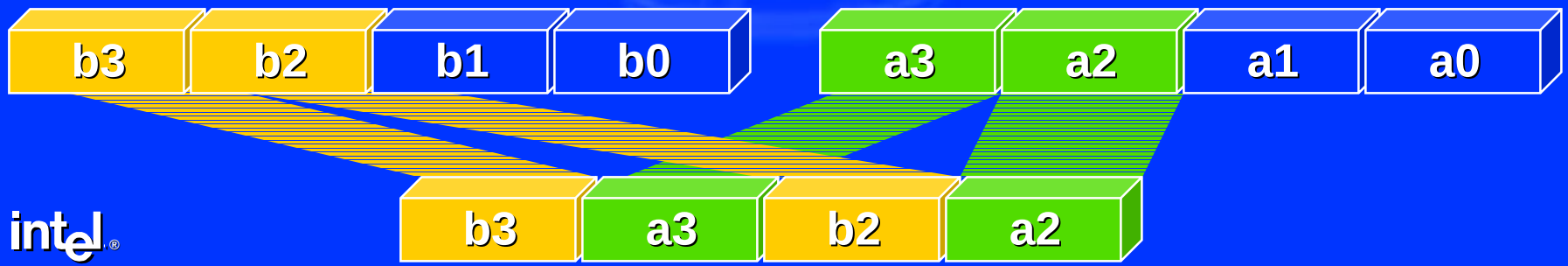
# Chief Unswizzler: Unpack

- Two lower(higher) elements from the first operand and two lo(hi) ones from the second are interleaved

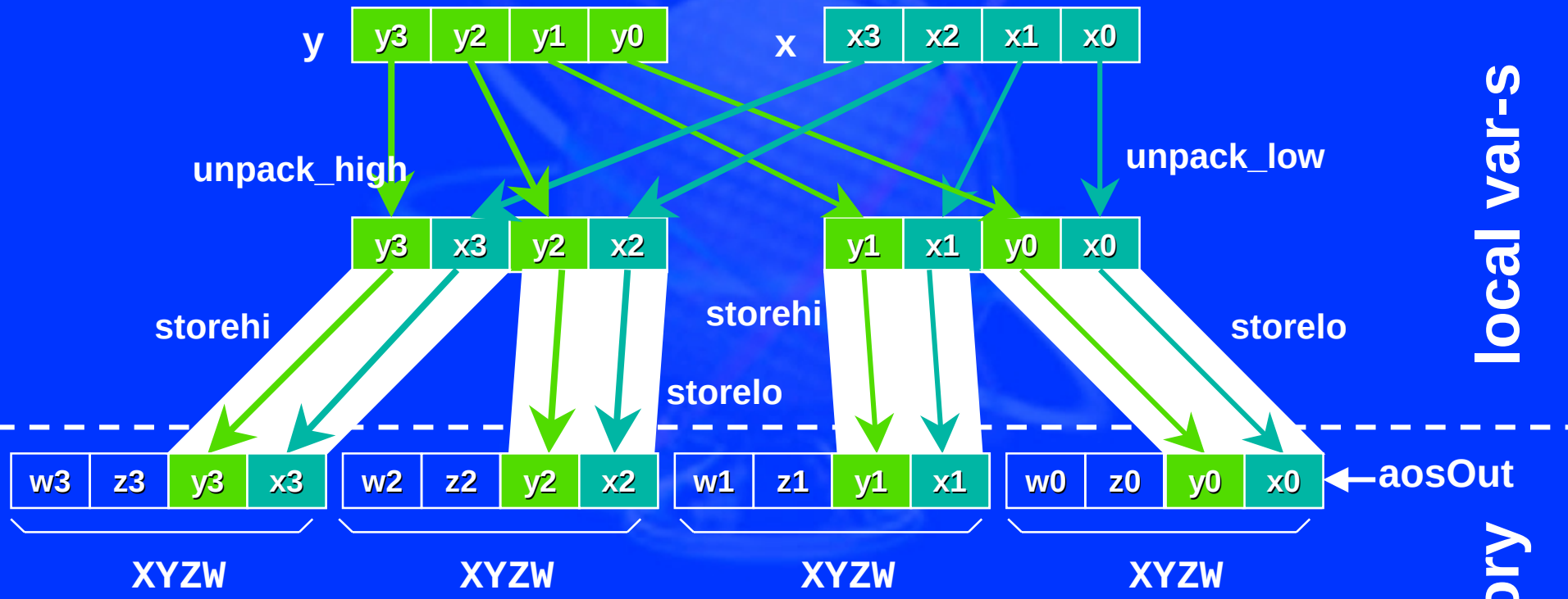
`unpack_low(a, b)`




`unpack_high(a, b)`

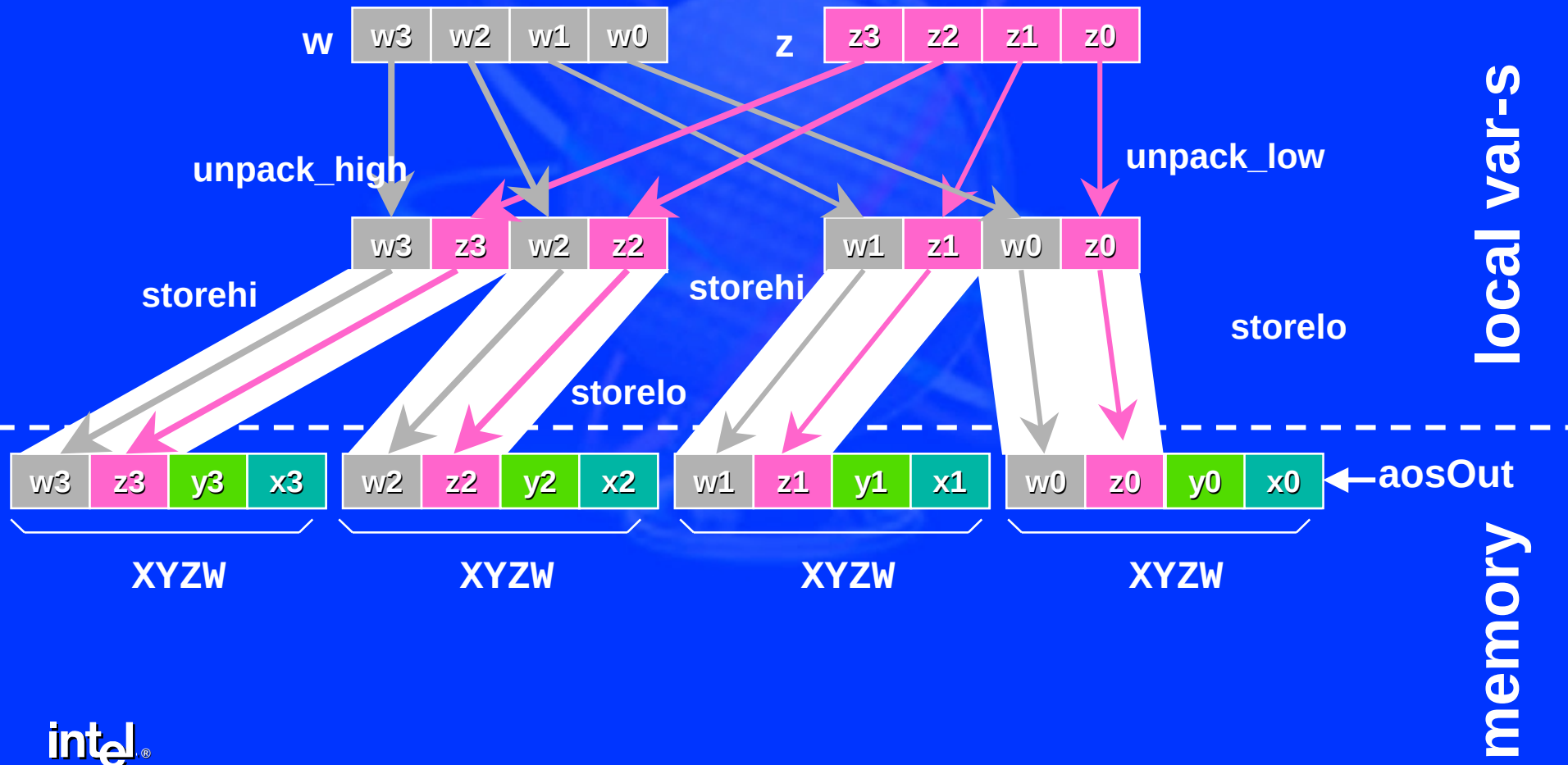


# Data Unswizzling with SIMD SoA to AoS

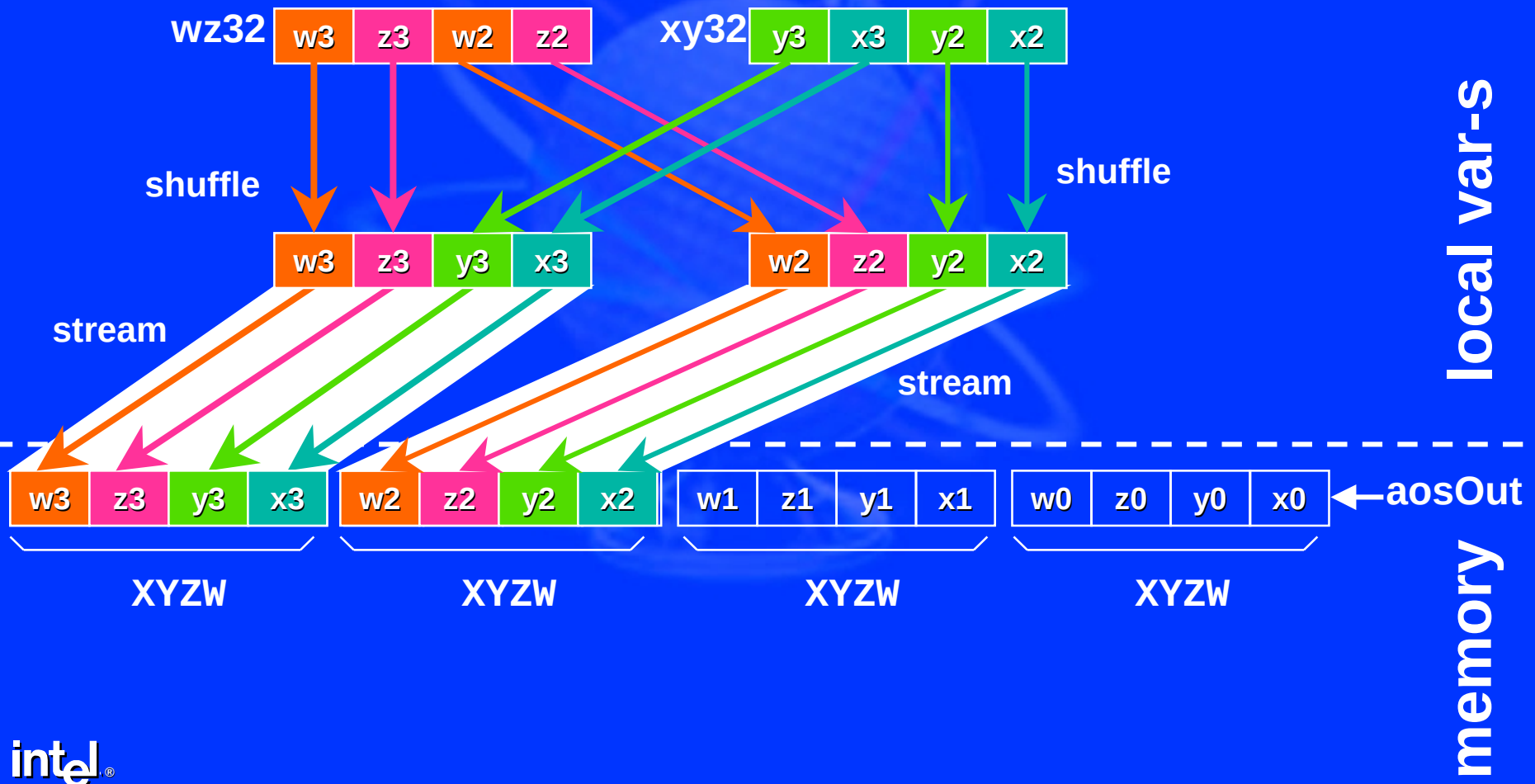



**Similar steps for z, w!**

# Data Unswizzling with SIMD SoA to AoS, Continued



# SoA to AoS with Streaming Store



# Data Restructuring Summary

- Use SIMD-friendly SoA or Hybrid structures whenever possible
- Use SSE/2 to swizzle SIMD-unfriendly structures before processing
- Use SSE/2 to store results of SIMD processing into SIMD-unfriendly structures (unswizzling)
- Look for more restructuring solutions in Bonus Foils!

intel®

# Agenda

- ✓ Exploiting Parallelism
- ✓ Data Restructuring
- ➢ Data Compression
- ● Conditional Code with SIMD
- ● Summary
- ● Bonus Foils

# Data Compression with Integers

- FP value inside a known range can be mapped into a compacter int value

$$i = i_{\min} + \left[ (f - f_{\min}) \cdot \frac{i_{\max} - i_{\min}}{f_{\max} - f_{\min}} \right]; \quad \text{for symmetric range: } i = \left[ f \cdot \frac{i_{range}}{f_{range}} \right]$$

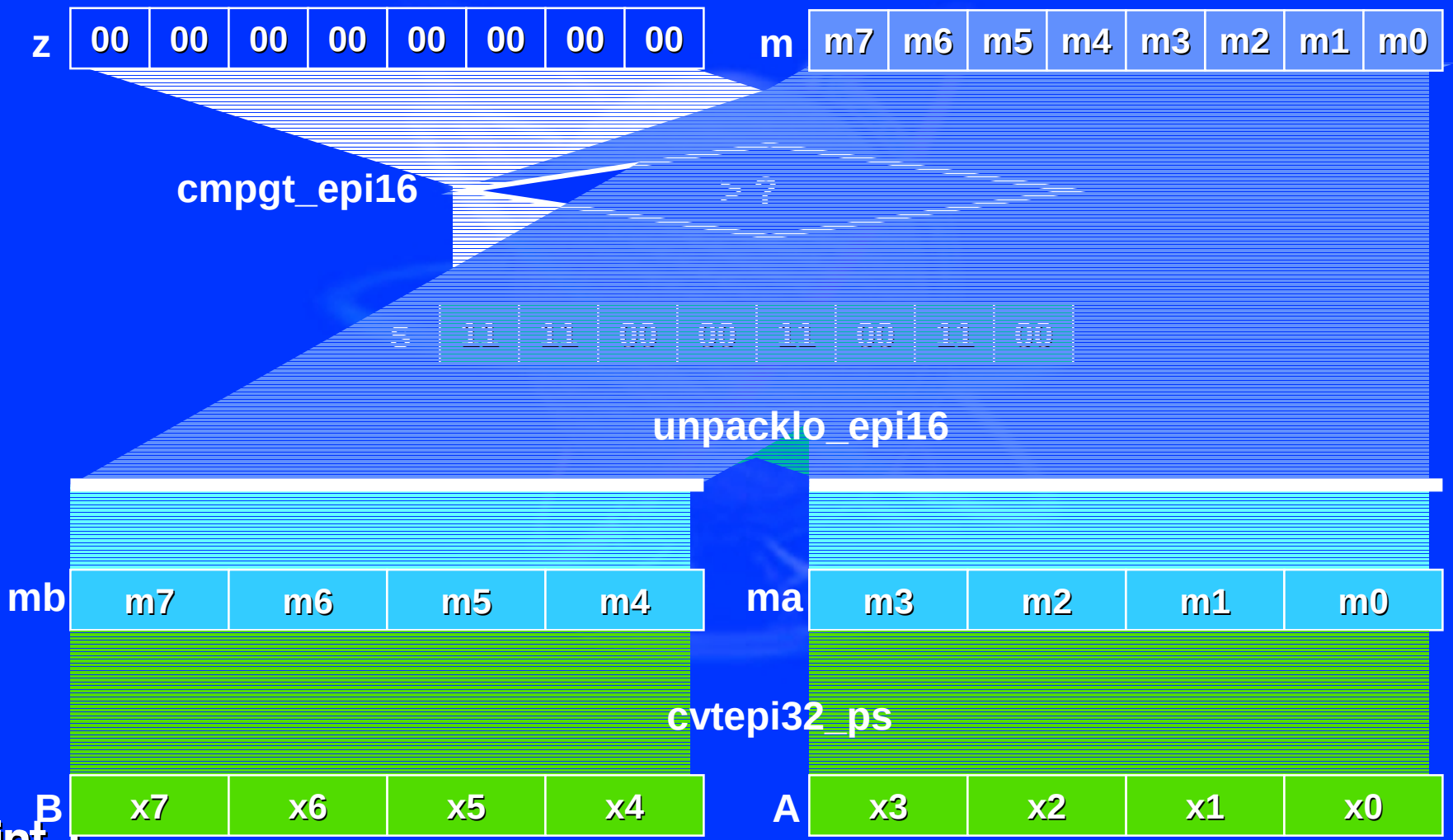
$$f = f_{\min} + (i - i_{\min}) \cdot \frac{f_{\max} - f_{\min}}{i_{\max} - i_{\min}}; \quad \text{for symmetric range: } f = i \cdot \frac{f_{range}}{i_{range}}$$

- Example: -1.0..+1.0 → -/+SHRT_MAX

```
short s; float f;
s = (short)round(f * SHRT_MAX);
f = float(s) * (1.0f / SHRT_MAX);
```

- Application examples: facet normals, lighting normals, landscape heights…

# SIMD Short → SIMD Float Conversion with SSE2



# Data Compression Summary

- Save memory traffic and cache space by [de]compressing data on-the-fly
- Use SSE / SSE2 for type conversions
- Swizzle short integer data before conversion – achieve wider parallelism

# Agenda

- ✓ Exploiting Parallelism
- ✓ Data Restructuring
- ✓ Data Compression
- ➢ Conditional Code with SIMD
- Summary
- Bonus Foils

# Conditions without Branches

```
R = (A < B)? C : D //remember: everything packed
```

# Conditions without Branches Code

```
// R = (A < B)? C : D

F32vec4 mask = cmplt(a, b);
r = (mask & c) | _mm_nand_ps(mask, d);

// OR, using F32vec4 friend function:
r = select_lt(a, b, c, d);
```

# Conditional Processing with SSE / SSE2

- Scalar

```
if (a < b)
    calculate c
    r = c
else
    calculate d
    r = d
```

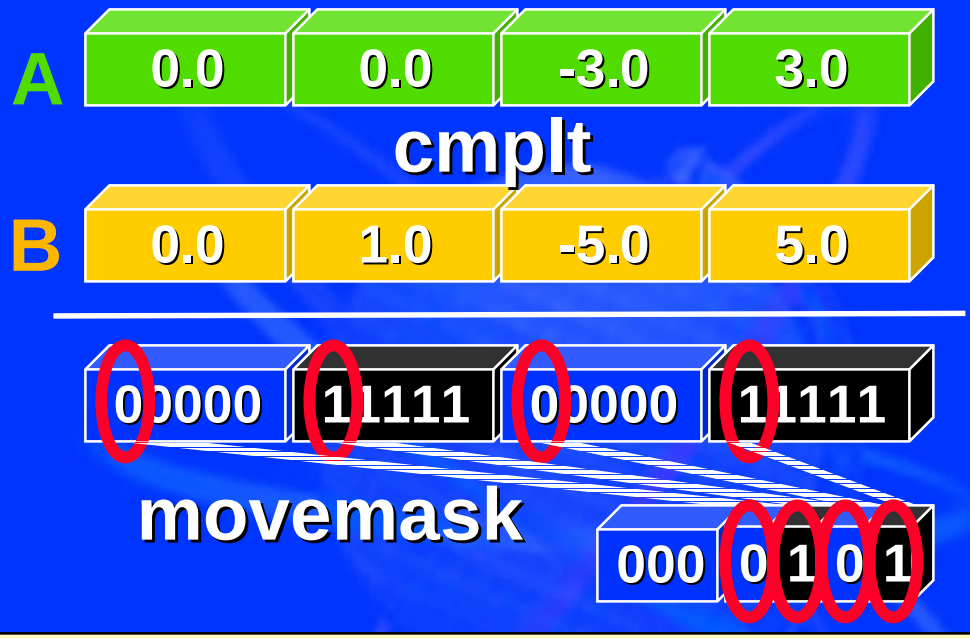
- SSE / SSE2

```
mask = cmplt(a, b)
calculate c
calculate d
r = (mask & c) |
      nand(mask, d)
```

**Utilize data-level and instruction-level parallelism!**

intel®

# Branch Hub: Movemask




```
F32vec4 mask = cmplt(a, b);
if (move_mask(mask)) {
 // do only if at least one is true
 // can be logic-conditional here
}
 case 15: //
}
```

One jump is better than many!

# Conditional Processing with SSE / SSE2, Movemask

- Scalar

```
if (a < b)
    calculate c
    r = c
else
    calculate d
    r = d
```

- SSE/SSE2, Movemask

```
mask = cmplt(a, b)
switch (move_mask(mask))
case 0xf:
    calculate c
    r = c
case 0x0:
    calculate d
    r = d
default:
    calculate c
    calculate d
    r = (mask&c)|nand(mask,d)
```

# SIMD for Conditional Code Summary

- You can successfully use SSE/SSE2 even with conditional, branchy code
- Replace branches with logic or computation
- Reduce total number of branches with movemask
- Look for more examples in Bonus Foils

# Agenda

- ✓ Exploiting Parallelism
- ✓ Data Restructuring
- ✓ Data Compression
- ✓ Conditional Code with SIMD
- ➢ Summary
- ● Bonus Foils

# What Is in Bonus Foils

- Using Automatic Vectorizer
  - Compiler can do SSE/SSE2 for you!
- More Conditional Code with SIMD
  - Abs function, flag accumulation ("clipping"), test passed counting…
- Applying SSE/SSE2 to Scalar Code
  - What if algorithm is inherently scalar? or there are no long data arrays?
  - Still get performance with SSE/SSE2!

# Summary: Call to Action

- Accelerate all your critical code with SSE / SSE2 processing
- Make your data SIMD-friendly
- Use SSE / SSE2 for on-the-fly data swizzling and [de]compression
- Use SSE / SSE2 comparisons & logic to replace conditional code
- Extend your own SSE/SSE2 Toolbox!

http://developer.intel.com/design/pentium4/
http://developer.intel.com/IDS

# Thank You!

# Happy Coding with SSE / SSE2!

# Bonus Foils

# Bonus Foils

- Using Automatic Vectorizer
- More Conditional Code with SIMD
- Applying SIMD to Scalar Code

# Using Intel Compiler's Automatic Vectorizer

- Now that SSE/SSE2 is so easy, the compiler can do it for you!
- Steps to using Automatic Vectorizer:

1. Understand for yourself how to SIMDize
2. Prepare and align the data structures
3. Provide hints such as unaliased pointers
4. Invoke Automatic Vectorizer
5. SIMDize remaining critical code with Vector Classes and Intrinsics

intel®

# Invoking Automatic Vectorizer

## -O2 -QaxW -Qvec_report3

- -O2 “optimize for speed”
  - standard Visual C++* Release build setting
- -QaxW “optimize using SSE and SSE2”
  - also invokes Automatic Vectorizer
  - auto-versions optimized code for compatibility
  - ignored by Microsoft* C++ compiler
- -Qvec_report3 “report on vectorization”
- See Intel Compiler documentation for more power options!

# Automatic Vectorizer in Action

```
void MbyV(float* xi, float* yi, float* zi, float* wi,
     float* restrict xo, float* restrict yo,
     float* restrict zo, float* restrict wo)
{
  __assume_aligned(xi, 16); ... // same for yi,zi,wi
  for (int i = 0; i < ARRAY_COUNT; i++) {
     float x = xi[i]; float y = yi[i];
     float z = zi[i]; float w = wi[i];
     wr = 1.0 / (x * matrix[3][0] + y * matrix[3][1] +
```

ClassView | FileView

```
--------------------Configuration: AoSSoA - Win32 Release--------------------
Compiling...
icl AoSSoA.cpp
C:\users\IdfSpring99\Lab1\AoSSoA.cpp(68) : (col. 2) remark: LOOP WAS VECTORIZED.

AoSSoA.obj - 0 error(s), 0 warning(s)
```

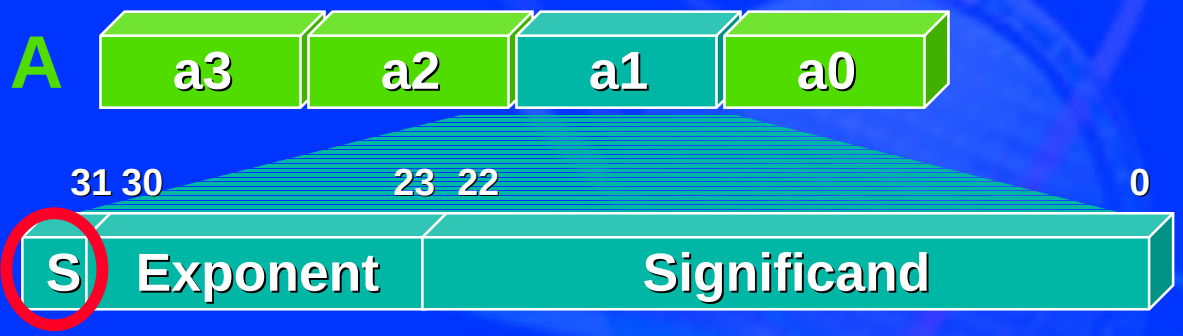
Build | Debug | Find in Files 1 | Fin

# Bonus Foils

- ✓ Using Automatic Vectorizer
- ➢ More Conditional Code with SIMD
- ● Applying SIMD to Scalar Code

# Implementing Abs with Logic

- Reminder: SIMD FP format




$$S = \begin{cases} 1, & \text{if negative} \\ 0, & \text{if non-neg} \end{cases}$$

- Code

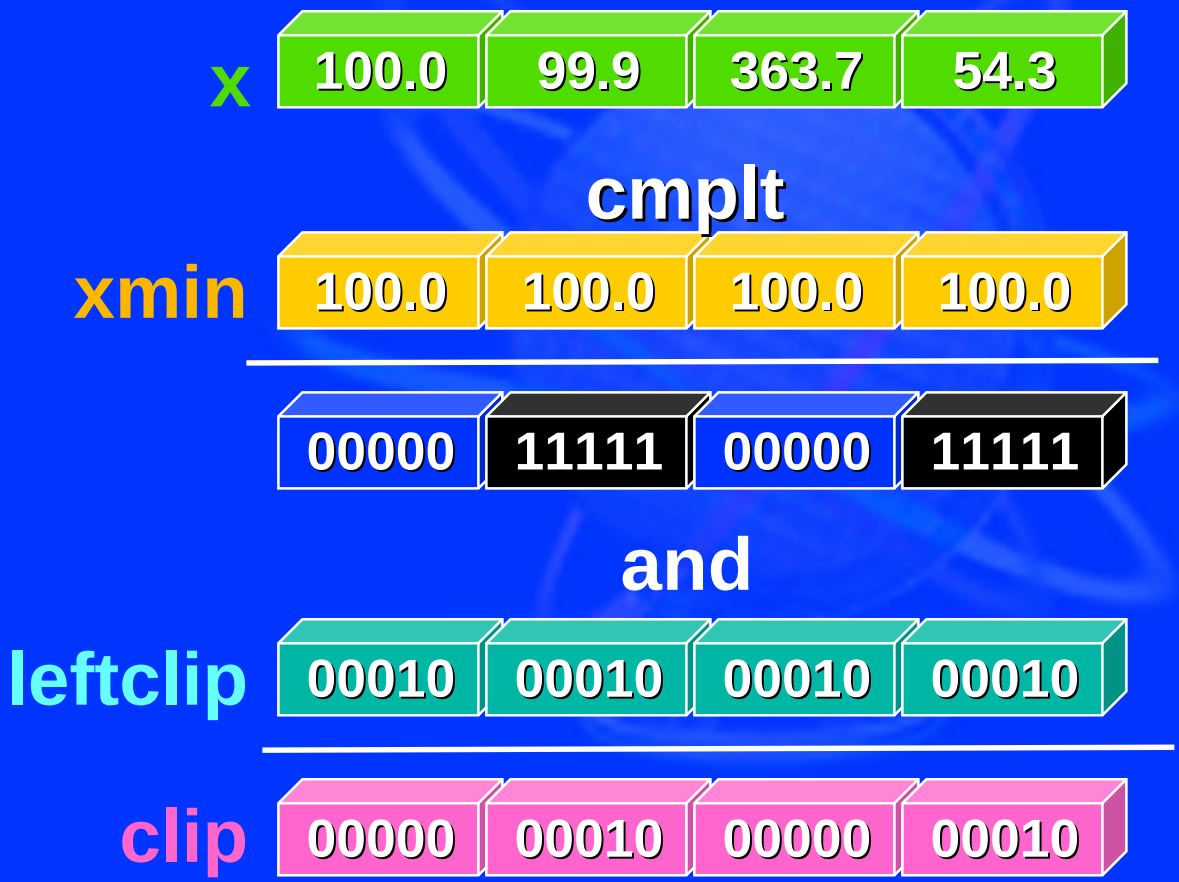
```
// r = abs(a)
CONST_INT32_PS(smask,
    ~(1<<31), ~(1<<31), ~(1<<31), ~(1<<31));
r = smask & a;
```

intel®

# Flag Accumulation: Original Scalar Code

```
char clip = 0;

if (v->x < xmin)
   clip |= LEFTCLIP;
else if (v->x > xmax)
   clip |= RIGHTCLIP;
if (v->y < ymin)
   clip |= TOPCLIP;
else if (v->y > ymax)
   clip |= BOTTOMCLIP;
```

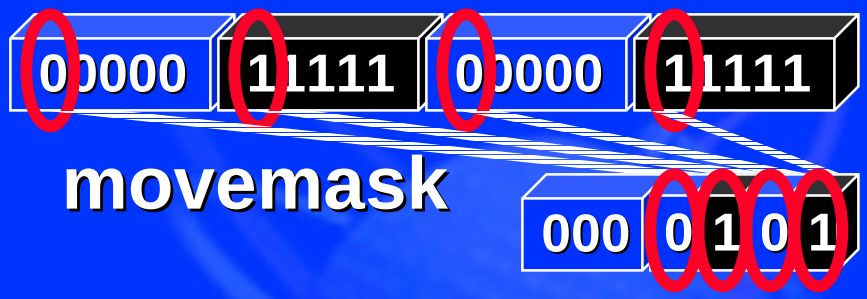
# Flag Accumulation with SSE / SSE2



# Flag Accumulation with SSE / SSE2 Code

```
DEFCONST_INT_PS(leftclip, LEFTCLIP);
... // DEFCONST for rightclip, topclip, botclip
F32vec4 clip, mask;
__m128i iclip;
unsigned uclip;

mask = cmplt(sx, ps_xmin);
clip = mask & leftclip;
mask = cmpgt(sx, ps_xmax);
clip |= mask & rightclip;
mask = cmplt(sy, ps_ymin);
clip |= mask & topclip;
mask = cmpgt(sy, ps_ymax);
clip |= mask & botclip;
// pack int32 → int8
iclip = (__m128i&)clip;                    // cast type
iclip = _mm_packs_epi32(iclip, iclip);     // pack 32 → 16
iclip = _mm_packus_epi16(iclip, iclip);    // pack 16 → 8
uclip = _mm_cvtsi128_si32(iclip);          // move to int
```

# Test Passed Counting




```
static const int bitcount[16] = {
    0, // 0 == 0000
    1, // 1 == 0001
    1, // 2 == 0010
    2, // 3 == 0011
    ....
    4  // 15 == 1111
};
F32vec4 mask = cmplt(a, b);
npassed = bitcount[move_mask(mask)];
```

# Bonus Foils

- ✓ Using Automatic Vectorizer
- ✓ More Conditional Code with SIMD
- ➢ Applying SIMD to Scalar Code

# Applying SIMD to Scalar Code

- SSE can be applicable inside a scalar algorithm without global parallelization
- Accelerate general processing with SSE operations
  - SSE registers are more efficient than x87
  - SSE divide, square root – rcp, rsqrt
  - type conversions – cvtsi2ss, cvt(t)ss2si…
  - comparisons – comiss, comisd
- Accelerate operations common to all heterogeneous components

# Is It Really-Really Scalar?

- In most cases, can easily load scalar data into SSE/SSE2 operands
  - Load four random 3-comp vectors:

```
void XYZToF32vec4(F32vec4& x, y, z, const XYZ* p0, p1, p2, p3)
{
  CONST_INT32_PS(m20, 0,~0,0,~0);  // mask for elements 2, 0
  F32vec4 a, b, c, d, e;

  a = _mm_loadu_ps(&p0->x);        // --,z0,y0,x0
  b = _mm_loadu_ps((&p1->x) - 1);  // z1,y1,x1,--
  c = (m20 & a) | andnot(m20, b);  // z1,z0,x1,x0
  b = (m20 & b) | andnot(m20, a);  // --,y1,y0,--

  a = _mm_loadu_ps(&p2->x);        // --,z2,y2,x2
  d = _mm_loadu_ps((&p3->x) - 1);  // z3,y3,x3,--
  e = (m20 & a) | andnot(m20, d);  // z3,z2,x3,x2
  d = (m20 & d) | andnot(m20, a);  // --,y3,y2,--

  x = _mm_movelh_ps(c, e);         // x3,x2,x1,x0
  z = _mm_movehl_ps(e, c);         // z3,z2,z1,z0
  y = _mm_shuffle_ps(b, d, _MM_SHUFFLE(2,1,2,1)); // y3,y2,y1,y0
}
```

# Avoiding SIMD Catches for Scalar Data

- Example: load XYZ vector as SSE operand
- Catch 1: Misalignment

```
F32vec4 v; XYZ* vec;
x = _mm_loadl_pi(&vec->x);
v = _mm_movelh_ps(x, _mm_load_ss(&vec->z));
```

  - loadlo, loadhi slow when not 8-byte aligned
- Catch 2: FP “Junk” data

```
x = _mm_loadu_ps(&vec->x);
```

  - Junk data leads to “special” values in math operations → slowdown!

# Loading XYZ Vector as SSE Operand, Good Way

```
F32vec4 v;
XYZ* vec;
v = _mm_loadu_ps(&vec->x);
v = v & mask210;
// OR:
// v = _mm_shuffle_ps(v, v,
//        _MM_SHUFFLE(2,2,1,0));
```

- One slow unaligned load, one logic
- Junk data masked out
- Aligned load would be much faster
- Data alignment is still important!

intel®

# Loading XYZ Vector as SSE Operand, Better Way

```
F32vec4 v, y, z;
XYZ* vec;
v = _mm_load_ss(&vec->x); // 0,0,0,x
y = _mm_load_ss(&vec->y); // 0,0,0,y
z = _mm_load_ss(&vec->z); // 0,0,0,z
v = _mm_movelh_ps(v, y); // 0,y,0,x
v = _mm_shuffle_ps(v, z, S(2,0,2,0));
```

- Three fast loads, two shuffles
- ~1.3x faster than non-aligned SIMD load
- ~2x slower than aligned SIMD

intel®

# SIMD Element Sumup

- Used widely in SIMD-for-Scalar code
- Requires two sequential shuffles

```
inline F32vec1 sumup(F32vec4 x)
{
 x += _mm_movehl_ps(x, x);
 ((F32vec1&)x) += _mm_shuffle_ps(x, x, S(3,2,1,1));
 return x;
}
```

# Parallel Element Sumups

- Four element sumups in parallel
- 1.5 independent shuffles per sumup

```
inline F32vec4 sumup(F32vec4 a, b, c, d)
{
 a = unpack_low(a, b) + unpack_high(a, b);
                        // b3+b1, a3+a1, b2+b0, a2+a0
 c = unpack_low(c, d) + unpack_high(c, d);
                        // d3+d1, c3+c1, d2+d0, c2+c0
 b = _mm_movelh_ps(a, c);
                        // d2+d0, c2+c0, b2+b0, a2+a0
 d = _mm_movehl_ps(c, a);
                        // d3+d1, c3+c1, b3+b1, a3+a1
 a = b + d;
 return a;
}
```

# Vector Normalize, SSE SIMD

- Element sumup considered earlier
- 5 shuffles, 2 multiplies
- Aligning data would speed it up!

```
F32vec4 v, s;
F32vec1 t;
v = _mm_loadu_ps(inVec);
v = v & mask210;

s = v * v;
t = sumup3(s);   // sum up 3 lower elements only
t = rsqrt_nr(t); // SSE scalar
v *= _mm_shuffle_ps(t, t, S(0,0,0,0));

_mm_storeu_ps(outVec, v);
```

# Vector Normalize, SSE Scalar

- 0 shuffles, 6 multiplies, reschedulable loads
- ~20% faster than unaligned SSE SIMD
- ~15% slower than aligned SSE SIMD

```
F32vec1 x, y, z, t;
x = _mm_load_ss(&inVec->x);
y = _mm_load_ss(&inVec->y);
z = _mm_load_ss(&inVec->z);
t = x * x + y * y + z * z;
t = rsqrt_nr(t); // SSE scalar
x *= t;
y *= t;
z *= t;
_mm_store_ss(&outVec->x, x);
_mm_store_ss(&outVec->y, y);
_mm_store_ss(&outVec->z, z);
```

# SSE SIMD or SSE Scalar?

- Depends on memory loads to processing operation ratio
- Aligned load / store are the fastest
- Homogeneous component processing is faster with packed operations
- Load / store of separate components is more efficient than unaligned SIMD load

# Matrix by Vector, Smart Scalar

# SIMD for Scalar Code Summary

- Inherently scalar algorithms also benefit from SSE / SSE2
- Aligning data is still important!
- Do not tolerate junk data elements
- Use SSE / SSE2 fast operations: reciprocal, compares, conversions

# SIMD Synonyms

- Nouns:
  - SSE = SSE / SSE2 = SSE2 = SIMD
  - "SIMD operand consists of {four} elements"
- Adjectives:
  - SIMD data = vectorized data = packed data
- Verbs:
  - SIMDize = vectorize = parallelize