Cell Broadband Engine
SIMD Programming

Lars Karlsson

May 5, 2009

A Word About SIMD...

There are two prevalent SIMD designs:

1. Vector instructions (e.g., Cell).
2. Synchronous execution of (scalar) units (e.g., GPUs).

The second type is more flexible from a programming viewpoint since individual units can branch arbitrarily. However, performance implications are more apparent in the first type since all units branch together and read/write contiguous data.

Vector Types

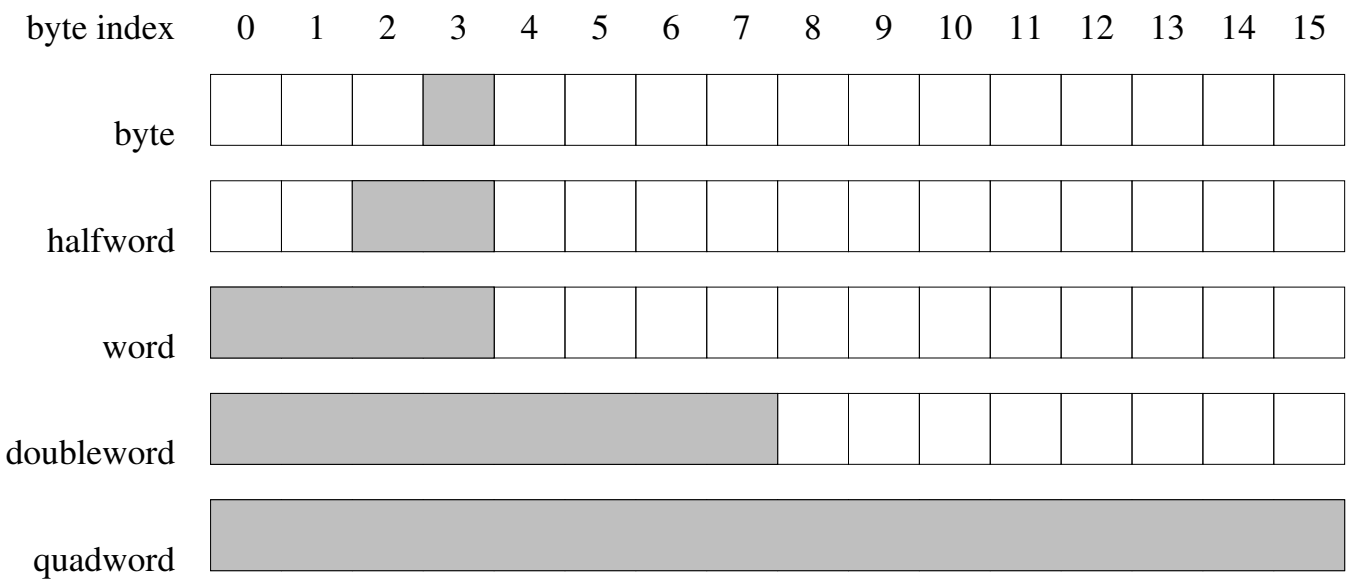
- The SPU has 128-bit registers, consisting of either
  - 16 char
    - `vector (signed|unsigned) char`
  - 8 halfword
    - `vector (signed|unsigned) short`
  - 4 word
    - `vector (signed|unsigned) int`
    - `vector float`
  - 2 doubleword
    - `vector (signed|unsigned) long long`
    - `vector double`
  - 1 quadword
    - `qword`
- Vector literals specified in C99 compound object literal format:

```
(vector float) {1.3f, 5.2f, -2.342f, 0.f}
```

- Vector components accessed using array index syntax:

```
vector float v = (vector float) {1.3f, 5.2f, -2.342f, 0.f};
v[1]; // Returns 5.2f
```

Preferred Scalar Slot



- When scalars are used or produced, the values are in the preferred scalar slot.

Vector Operators

- Arithmetic operators are overloaded on vector types:

```
vc = va + vb; // element-wise addition
vc = va - vb; // element-wise subtraction
vc = va * vb; // element-wise multiplication
vc = va / vb; // element-wise division
```

- Also:
  - relational operators
  - unary operators
  - bit-wise logical operators
- Relational operators give a scalar result which is nonzero iff the relational operator give a nonzero result for each element.
- Pointers to scalars and vectors can be cast back and forth but vector addresses are assumed to be 16-byte aligned.

```
int v[8] __attribute__((aligned(16))) = {1,2,3,4,5,6,7,8};
*(vector signed int*) &v[2]; // returns {1,2,3,4}, NOT {3,4,5,6}
```

Scalars and Vectors

- Initializer:
  `vector signed int v = {1,2,3,4}`
- Replicate scalar a in vector d:
  `d = spu_splats(a)`
- Get specific element from a:
  `d = spu_extract(a, element)`
  Better yet, use [] operator:
  `d = a[element]`
- Place scalar a in specific element of b:
  `d = spu_insert(a, b, element)`
  Better to use [] operator:
  `b[element] = a`
- Turn scalar a into vector d by placing it in specified element:
  `d = spu_promote(a, element)`

Shift and Rotate

- Rotate elements in a left by count bits:
  `d = spu_rl(a, count)`
- Logical shift elements in a right by -count bits:
  `d = spu_rlmask(a, count)`
- Rotate register a left by count bytes:
  `d = spu_rlqwbyte(a, count)`
- Logical shift register a right by -count bytes:
  `d = spu_rlmaskqwbyte(a, count)`
- *Many more...*

Select Bits

- Select bits from a or b depending on pattern:
  `d = spu_sel(a, b, pattern)`
  If bit is zero: take from a, if one: take from b.
- Form select byte pattern:
  `d = spu_maskb(a)`
  The 16 least significant bits of a are replicated 8 times.
- Form select halfword pattern:
  `d = spu_maskh(a)`
  The 8 least significant bits of a are replicated 16 times.
- Form select word pattern:
  `d = spu_maskw(a)`
  The 4 least significant bits of a are replicated 32 times.

Vector Compares

These operations compare vectors elementwise and return a bitmask for use with `spu_sel`.

- Compare elements for equality
  `mask = spu_cmpeq(a, b)`
- Compare if element in a is greater than corresponding element in b
  `mask = spu_cmpgt(a, b)`

Select Bits: Example

- Set all negative integers in a to zero:

```
vector signed int a, b;
vector unsigned int mask;

// Negate all elements in a:
b = -a;

// Find negative elements in a (which are positive in b):
mask = spu_cmpgt(b, 0);

// Select result from a and 0:
a = spu_sel(a, spu_splats((signed int) 0), mask);
```

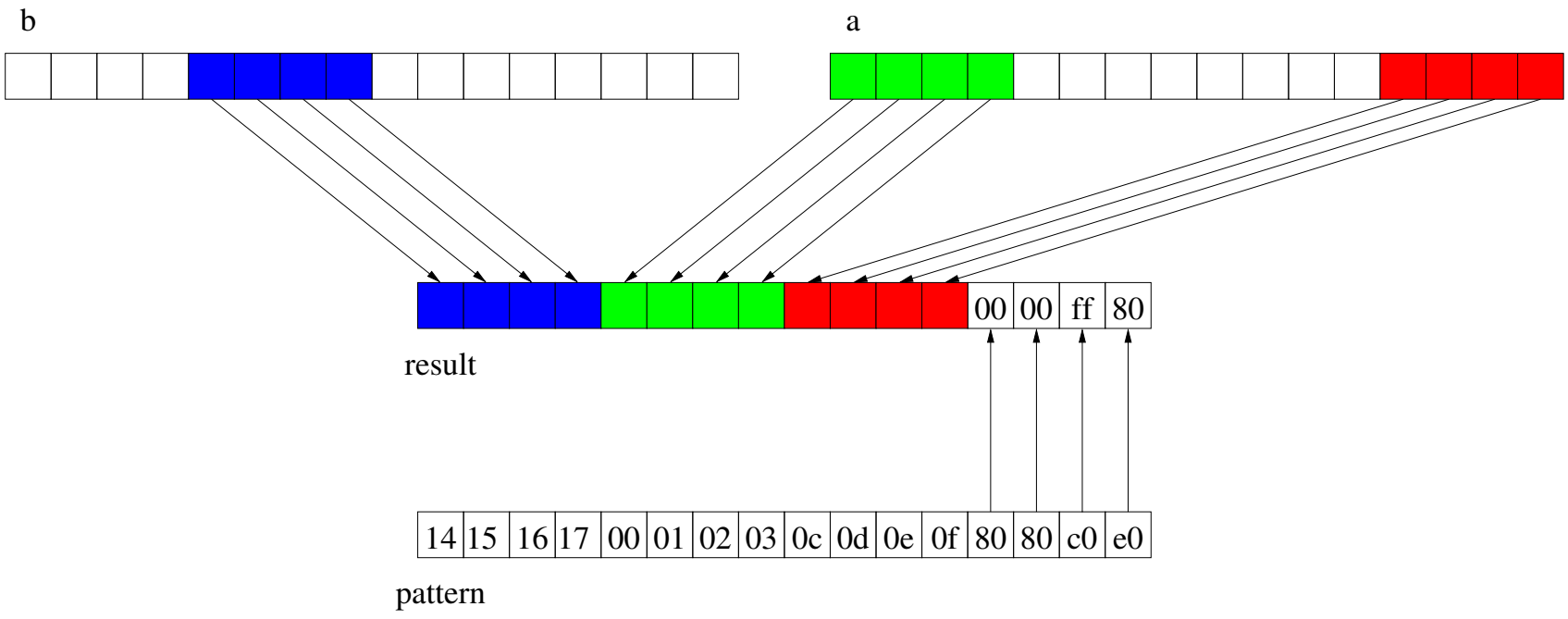
Shuffle

- Shuffle produces output by picking bytes from a, b, or by selecting from a set of predefined constant bytes.
- Special bytes:

pattern	result
10xxxxxx	0x00
110xxxxx	0xff
111xxxxx	0x80

- Otherwise, 5 least significant bits selects among the 32 bytes of a and b.
- Examples:
  0x0a – selects byte 10 from a
  0x16 – selects byte 6 from b
- In general: bit 4 selects a (0) or b (1) while bits 0..3 selects a byte in the selected register.

Shuffle: Example



Shuffle: 4×4 Transpose

$$A = \left[\begin{array}{cccc} a1 & a2 & a3 & a4 \\ \hline b1 & b2 & b3 & b4 \\ \hline c1 & c2 & c3 & c4 \\ \hline d1 & d2 & d3 & d4 \end{array}\right]$$

$$B = \left[\begin{array}{cccc} a1 & c1 & a2 & c2 \\ \hline a3 & c3 & a4 & c4 \\ \hline b1 & d1 & b3 & d3 \\ \hline b3 & d3 & b4 & d4 \end{array}\right]$$

$$A^T = \left[\begin{array}{cccc} a1 & b1 & c1 & d1 \\ \hline a2 & b2 & c2 & d2 \\ \hline a3 & b3 & c3 & d3 \\ \hline a4 & b4 & c4 & d4 \end{array}\right]$$

Shuffle: 4×4 Transpose

```
vector float r, s, t, u;
vector unsigned char pat_hi = {0x00, 0x01, 0x02, 0x03,  // a1
                               0x10, 0x11, 0x12, 0x13,  // b1
                               0x04, 0x05, 0x06, 0x07,  // a2
                               0x14, 0x15, 0x16, 0x17}; // b2
vector unsigned char pat_lo = {0x08, 0x09, 0x0a, 0x0b,  // a3
                               0x18, 0x19, 0x1a, 0x1b,  // b3
                               0x0c, 0x0d, 0x0e, 0x0f,  // a4
                               0x1c, 0x1d, 0x1e, 0x1f}; // b4
// First step.
r = spu_shuffle(a, c, pat_hi);
s = spu_shuffle(a, c, pat_lo);
t = spu_shuffle(b, d, pat_hi);
u = spu_shuffle(b, d, pat_lo);
// Second step.
a = spu_shuffle(r, t, pat_hi);
b = spu_shuffle(r, t, pat_lo);
c = spu_shuffle(s, u, pat_hi);
d = spu_shuffle(s, u, pat_lo);
```

Shuffle: 4×4 Transpose

$$P = \left[\begin{array}{cccc|cccc|cccc|cccc}
1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & 1 & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & 1 & . & . & . & . & . & . \\
\hline
. & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & 1 & . & . & . & . \\
\hline
. & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . & . \\
\hline
. & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . & 1 & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & 1
\end{array}\right]^2$$

Arithmetic

- Add: $d \leftarrow a + b$
  `d = spu_add(a, b)`
- Subtract: $d \leftarrow a - b$
  `d = spu_sub(a, b)`
- Multiply: $d \leftarrow a \times b$
  `d = spu_mul(a, b)`
- Fused multiply and add: $d \leftarrow a \times b + c$
  `d = spu_madd(a, b, c)`
- Negative fused multiply and add: $d \leftarrow -(a \times b + c)$
  `d = spu_nmadd(a, b, c)`
- Fused multiply and subtract: $d \leftarrow a \times b - c$
  `d = spu_msub(a, b, c)`
- Negative fused multiply and subtract: $d \leftarrow -(a \times b - c)$
  `d = spu_nmsub(a, b, c)`

Arithmetic: Reciprocal

- The reciprocal operation ($d \leftarrow \frac{1}{a}$) is not supported in hardware.
- Reciprocal estimate instruction followed by one iteration of Newton-Raphson. Let r be estimate of reciprocal $1/x$.

$$f(r) = \frac{1}{r} - x \quad \text{and} \quad f'(r) = -\frac{1}{r^2}$$

$$r_{k+1} = r_k - \frac{f(r_k)}{f'(r_k)} = r_k - \frac{\frac{1}{r_k} - x}{-\frac{1}{r_k^2}} = r_k + r_k - x r_k^2 = r_k(2 - x r_k)$$

- $f(r) = 0 \Rightarrow r = \frac{1}{x}$.
- Compiler generates code from overloaded vector operator `d = 1 / a`
- Similar technique used for square root and reciprocal square root.

Integer Arithmetic

32-bit integer arithmetic not supported explicitly in hardware (only 16-bit multiplier). Therefore, follow these guidelines:

- Make array-element size a power of 2 to avoid multiplies when indexing.
- If operands are less than 16 bits in size, cast to `unsigned short` to take advantage of hardware multiplier.
- Always cast constants since they are implicit `int` (32-bit).

Misaligned Load

```
vector float misaligned_load(vector float *ptr) {
   vector float qw0, qw1;
   int shift;

   qw0 = *ptr; // load left part of result
   qw1 = *(ptr + 1); // load right part of result
   shift = (unsigned) ptr & 15; // offset of result in qw0
   return spu_or( // combine left and right part
            spu_slqwbyte(qw0, shift), // align left part
            spu_rlmaskqwbyte(qw1, shift-16)); // align right part
}
```

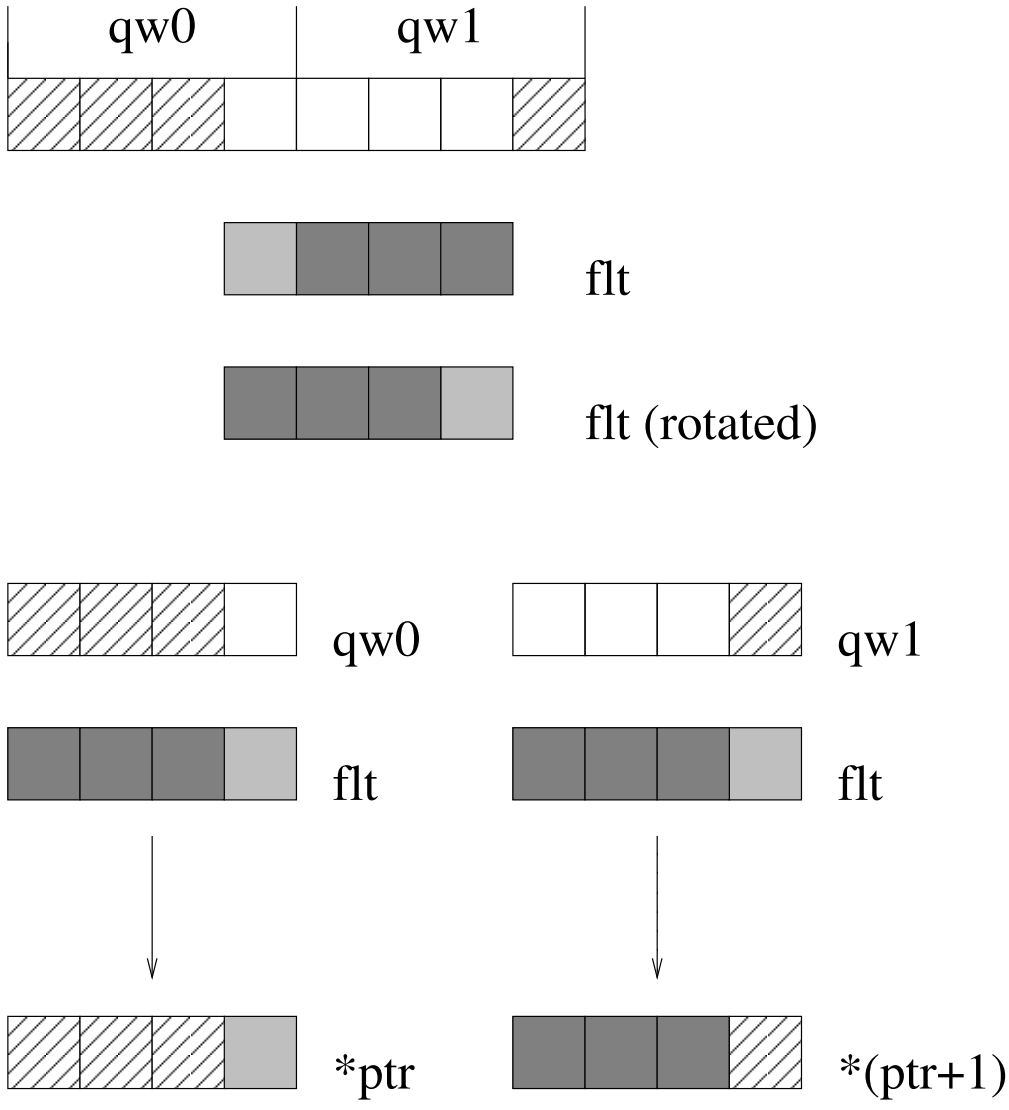
1. Shift left partial result to beginning of register.
2. Shift right partial result to end of register.
3. Combine partial results.

Misaligned Store

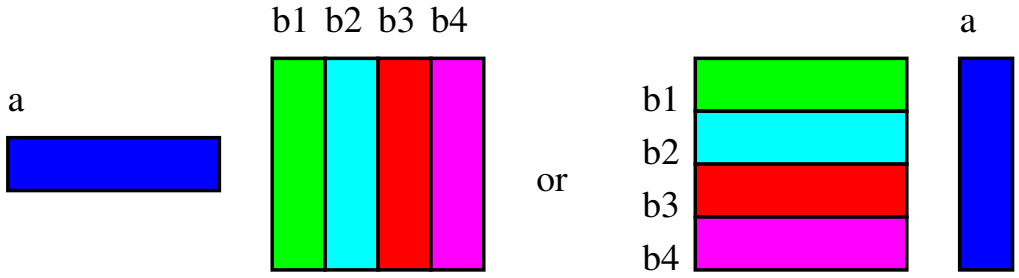
```
void misaligned_store(vector float flt, vector float *ptr) {
   vector float qw0, qw1;
   vector unsigned char mask;
   int shift;

   qw0 = *ptr; // load left part of destination
   qw1 = *(ptr + 1); // load right part of destination
   shift = (unsigned) ptr & 15; // offset of destination in qw0
   mask = spu_rlmaskqwbyte(spu_splats((unsigned char) 0xff), -shift);
   flt = spu_rlqwbyte(flt, -shift);
   *ptr = spu_sel(qw0, flt, mask);
   *(ptr + 1) = spu_sel(flt, qw1, mask);
}
```

1. Construct bitmask with zeros followed by ones.
2. Rotate input left by 0 or 16 - shift bytes.
3. Combine old with new data and store the two quadwords.

Misaligned Store: Illustration

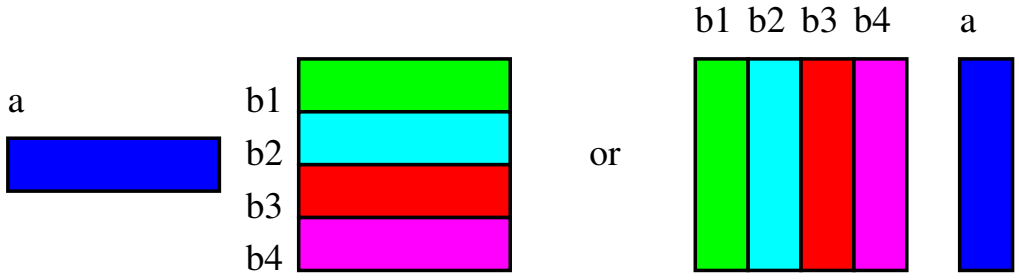


Matrix Vector Product (1, short)




```
// Form all the products in the four dot products.
t1 = a * b1;
t2 = a * b2;
t3 = a * b3;
t4 = a * b4;

// Transpose to align data.
TRANSPOSE_4x4(t1, t2, t3, t4);

// Reduce (sum terms in dot products).
c = t1 + t2 + t3 + t4;
```

Matrix Vector Product (2, short)




```
// Form all the products in the four dot products.
t1 = spu_splats(a[0]) * b1;
t2 = spu_splats(a[1]) * b2;
t3 = spu_splats(a[2]) * b3;
t4 = spu_splats(a[3]) * b4;

// Data is already aligned: reduce.
c = t1 + t2 + t3 + t4;
```

Matrix Vector Product (1, long)

```
// First block:
t1 = a * b1;
t2 = a * b2;
t3 = a * b3;
t4 = a * b4;

// Loop through remaining blocks.
for( int i = ...; i < ...; i++ ) {
   // Load a and b1..b4
   // Update t1..t4
   t1 += a * b1;
   t2 += a * b2;
   t3 += a * b3;
   t4 += a * b4;
}

// Transpose to align data.
TRANSPOSE_4x4(t1, t2, t3, t4);

// Reduce (sum terms in dot products).
c = t1 + t2 + t3 + t4;
```

Matrix Vector Product (2, long)

```
// First block:
t1 = spu_splats(a[0]) * b1;
t2 = spu_splats(a[1]) * b2;
t3 = spu_splats(a[2]) * b3;
t4 = spu_splats(a[3]) * b4;

// Loop through remaining blocks.
for( int i = ...; i < ...; i++ ) {
   // Load a and b1..b4
   // Update t1..t4
   t1 += spu_splats(a[0]) * b1;
   t2 += spu_splats(a[1]) * b2;
   t3 += spu_splats(a[2]) * b3;
   t4 += spu_splats(a[3]) * b4;
}

// Data is already aligned: reduce.
c = t1 + t2 + t3 + t4;
```

Array-of-Structure (AoS)

```
// Structure
struct el {
   vector float p; // Position (px, py, pz, -)
   vector float v; // Velocity (vx, vy, vz, -)
};

// Array-of-Structure
struct el ar[256];

// Integrate
for( int i = 0; i < 256; i++ ) {
   ar[i].p += ar[i].v * dt; // Wastes 1/4 of computations
}
```

Structure-of-Arrays (SoA)

```
// Structure
struct ar {
   vector float px[64], py[64], pz[64];
   vector float vx[64], vy[64], vz[64];
};

// Structure-of-Arrays
struct ar ar;

// Integrate (unrolled four times and vectorized)
for( int i = 0; i < 64; i++ ) {
   ar.px[i] += ar.vx[i] * dt; // Four iterations at once
   ar.py[i] += ar.vy[i] * dt; // No wasted operations
   ar.pz[i] += ar.vz[i] * dt;
}
```

On-the-fly Conversion between AoS and SoA

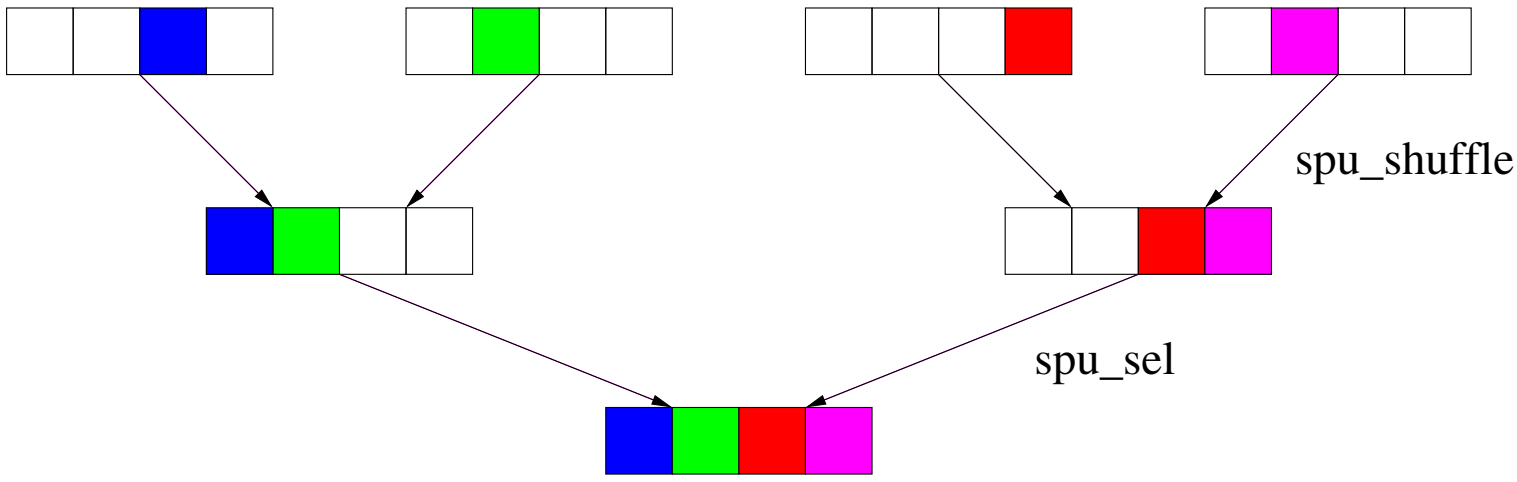
```
for( int i = 0; i < 256; i += 4 ) {
   // Load.
   p0 = ar[i+0].p;   p1 = ar[i+1].p;
   p2 = ar[i+2].p;   p3 = ar[i+3].p;
   v0 = ar[i+0].v;   v1 = ar[i+1].v;
   v2 = ar[i+2].v;   v3 = ar[i+3].v;
   TRANSPOSE_4x4(p0, p1, p2, p3);
   TRANSPOSE_4x4(v0, v1, v2, v3);
   // Compute in SoA form.
   p0 += v0 * dt; // px += vx * dt
   p1 += v1 * dt; // py += vy * dt
   p2 += v2 * dt; // pz += vz * dt
   // Store.
   TRANSPOSE_4x4(p0, p1, p2, p3);
   ar[i+0].p = p0;   ar[i+1].p = p1;
   ar[i+2].p = p2;   ar[i+3].p = p3;
}
```

On-the-fly Conversion between AoS and SoA

$$\left(\begin{array}{cccc} x_1 & y_1 & z_1 & - \\ \hline x_2 & y_2 & z_2 & - \\ \hline x_3 & y_3 & z_3 & - \\ \hline x_4 & y_4 & z_4 & - \end{array}\right) \longrightarrow \left(\begin{array}{c|c|c|c} x_1 & y_1 & z_1 & - \\ x_2 & y_2 & z_2 & - \\ x_3 & y_3 & z_3 & - \\ x_4 & y_4 & z_4 & - \end{array}\right)$$

- On the left: `p0` to `p3` contain the positions of four particles.
- On the right: after transposition, `p0` to `p2` contain the `x`, `y`, and `z` components of four particles each. `p3` contains all the garbage.

8-bit Table Lookup




```
vector unsigned char tbl[4];
vector unsigned char tbl0_1, tbl2_3, result;
vector unsigned char idx; // 0..63 (table indices)
vector unsigned char lsb3_7, bit2;

lsb3_7 = spu_and(idx, 0x1f); // shuffle indices
tbl0_1 = spu_shuffle(tbl[0], tbl[1], lsb3_7);
tbl2_3 = spu_shuffle(tbl[2], tbl[3], lsb3_7);
bit2 = spu_cmpeq(spu_and(idx, 0x20), 0x20);
result = spu_sel(tbl0_1, tbl2_3, bit2);
```