



# Advanced SIMD Programming

## Use cases in ARM NEON

Sorbonne Université – Master SESI – MU5IN60 – Parallel Programming

Adrien CASSAGNE

October 2, 2023



## Source of Inspiration

Acknowledgment

This document of is inspired by the excellent slides of Pr. Lionel LACASSAGNE (Sorbonne University) for the same class.

*Special thanks to him for sharing its work!*



# Table of Contents

## 1 Shuffle & Permutation

▶ Shuffle & Permutation

▶ Reduction

▶ Fixed-Point

▶ Mixed-Precision



# Pattern “left-right”

1 Shuffle & Permutation

- Purpose
  - Extract unaligned register from two aligned registers
  - Reduce the number of loads
- Scalar example
  - Let us suppose we have:  $a = 0x1234$ ,  $b = 0x5678$ ,  $c = 0x9abc$
  - We want:  $l=0x4567$  and  $r = 0x6789$
  - With bit shifts:  $l = (a \ll (3*4)) \mid (b \gg (1*4))$
  - With bit shifts:  $r = (b \ll (1*4)) \mid (b \gg (3*4))$
  - Tip: the sum of the shifts has to be equal to the number of digits (here  $4 \times 4$ )
- In NEON SIMD
  - `vext` (*EXTRACT*)



# Pattern “left-right” – vext

## 1 Shuffle & Permutation

- Principle

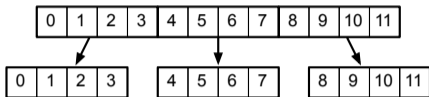
- `vext(a, b, n)`: `n` lanes from `b` and `card - n` lanes from `a`
- `n` is the number of lanes (or elements) and NOT the number of bytes

```
1 uint8x8_t a8 = (uint8x8_t) {0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7}; // [0 1 2 3 4 5 6 7]
2 uint8x8_t b8 = (uint8x8_t) {0x8, 0x9, 0xA, 0xB, 0xC, 0xD, 0xE, 0xF}; // [8 9 A B C D E F]
3 uint8x8_t c8 = vext_u8(a8, b8, 1); // [1 2 3 4 5 6 7 8]
4         c8 = vext_u8(a8, b8, 2); // [2 3 4 5 6 7 8 9]
5         c8 = vext_u8(a8, b8, 7); // [7 8 9 A B C D E]
6
7 uint16x4_t a16 = (uint16x4_t) {0, 1, 2, 3}; // [0 1 2 3]
8 uint16x4_t b16 = (uint16x4_t) {4, 5, 6, 7}; // [4 5 6 7]
9 uint16x4_t c16 = vext_u16(a16, b16, 1); // [1 2 3 4]
10         c16 = vext_u16(a16, b16, 2); // [2 3 4 5]
11         c16 = vext_u16(a16, b16, 3); // [3 4 5 6]
```



# Pattern “left-right” – 1D Stencil – Part 1

## 1 Shuffle & Permutation



- Stencil (or convolution, or tensor ;-))
  - Sum of 3 points:  
`for(int i=0; i<n; i++) Y[i]=X[i-1]+X[i]+X[i+1];`
  - We have:  $Y[4]=3+4+5$ ,  $Y[5]=4+5+6$ ,  $Y[6]=5+6+7$  and  $Y[7]=6+7+8$

```
1 float32x4_t a = vld1q_f32(&T[i - 1]); // [ 0 1 2 3]
2 float32x4_t b = vld1q_f32(&T[i   ]); // [ 4 5 6 7]
3 float32x4_t c = vld1q_f32(&T[i + 1]); // [ 8 9 A B]
4 float32x4_t y = vaddq_f32(vaddq_f32(a, b), c); // [0+4+8 1+5+9 2+6+A 3+7+B] <- not what we want
```

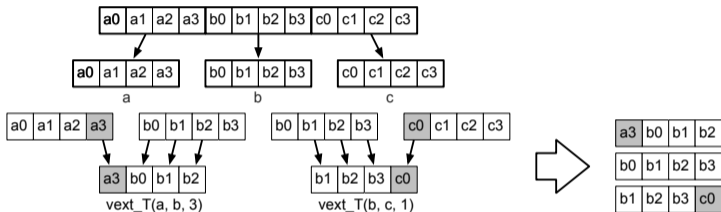
- Arithmetic operations are **vertical**
  - Additions are performed between SIMD registers



# Pattern “left-right” – 1D Stencil – Part 2

1 Shuffle & Permutation

- Sum of 3 points in SIMD
  - Compute unaligned registers
  - Add left, center and right registers

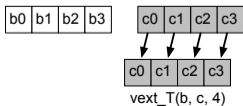
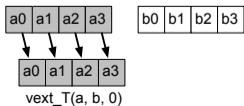
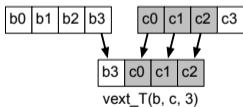
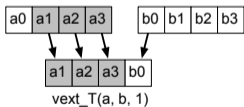
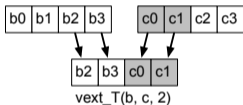
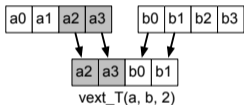
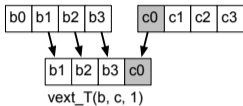
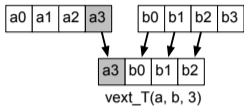


```
1 float32x4_t a = vld1q_f32(&T[i - 4]); // a = [0 1 2 3]
2 float32x4_t b = vld1q_f32(&T[i    ]); // b = [4 5 6 7]
3 float32x4_t c = vld1q_f32(&T[i + 4]); // c = [8 9 a b]
4
5 float32x4_t l = vextq_f32(a, b, 3); // l = [ 3  4  5  6]
6 // b = [ 4  5  6  7]
7 float32x4_t r = vextq_f32(b, c, 1); // r = [ 5  6  7  8]
8 float32x4_t y = vaddq_f32(vaddq_f32(a, b), c); // y = [3+4+5 4+5+6 5+6+7 6+7+8]
9
10 vst1q_f32(&T2[i], y); // store the results in an other array T2
```



# Pattern “left-right” – 1D Stencil – Conclusion

## 1 Shuffle & Permutation



- In scalar, for a stencil of size  $k$ 
  - $k - 1$  ADD +  $k$  LOAD + 1 STORE
  - AI (arithm. intensity) =  $(k - 1)/(k + 1) < 1$
- In SIMD, for a stencil of size  $k$ 
  - $k - 1$  ADD + 3 LOAD + 1 STORE
  - Number of LOAD is constant (if  $k \leq 2 \times card + 1$ )
  - AI (arithm. intensity) =  $(k - 1)/(3 + 1)$
  - **SIMD efficiency increases with  $k$**
- Limits for 3 LOAD (128-bit register length)
  - 32-bit:  $card = 4 \Rightarrow k = 9$ , max. gain= $\times 2.5$
  - 16-bit:  $card = 8 \Rightarrow k = 17$ , max. gain= $\times 4.5$
  - 8-bit:  $card = 16 \Rightarrow k = 33$ , max. gain= $\times 8.5$

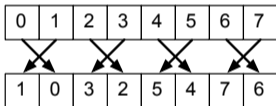




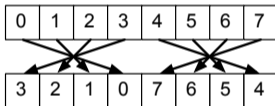
# Reverse Lanes Order – vrev – Part 1

1 Shuffle & Permutation

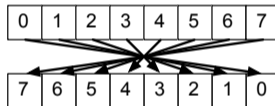
- Reverse the order of elements in 8-, 16- or 32-bit steps
- Use case: *Butterfly* for FFT



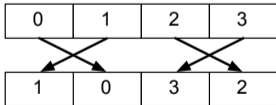
vrev16\_8



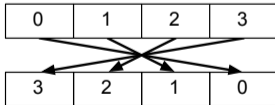
vrev32\_8



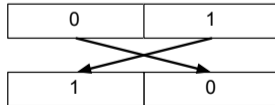
vrev64\_8



vrev32\_16



vrev64\_16



vrev64\_32

vrev (64-bit registers)



## Reverse Lanes Order – vrev – Part 2

### 1 Shuffle & Permutation

- 3 instructions:
  - vrev16: to reverse two 8-bit packets
  - vrev32: to reverse packets of four 8-bit or two 16-bit
  - vrev64: to reverse packets of eight 8-bit, four 16-bit or two 32-bit
- Example

```
1 uint8x8_t v8 = (uint8x8_t) {0, 1, 2, 3, 4, 5, 6, 7}; // [0 1 2 3 4 5 6 7]
2 uint8x8_t v8_r16 = vrev16_u8(v8); // [1 0 3 2 5 4 7 6]
3 uint8x8_t v8_r32 = vrev32_u8(v8); // [3 2 1 0 7 6 5 4]
4 uint8x8_t v8_r64 = vrev64_u8(v8); // [7 6 5 4 3 2 1 0]
5
6 uint16x4_t v16 = (uint16x4_t) {0, 1, 2, 3}; // [0 1 2 3]
7 uint16x4_t v16_r32 = vrev32_u16(v16); // [1 0 3 2]
8 uint16x4_t v16_r64 = vrev64_u16(v16); // [3 2 1 0]
9
10 uint32x2_t v32 = (uint32x2_t) {0, 1}; // [0 1]
11 uint32x2_t v32_r64 = vrev64_u32(v32); // [1 0]
```



# Reverse Lanes Order – vrev64\_8

## 1 Shuffle & Permutation

- Scalar code equivalent to the vrev64\_8 instruction

```
1 uint64_t u64 = 0x0123456789abcdef;
2 uint8_t a0 = u64 & 0xFF; u64 = u64 >> 8; // 0xef
3 uint8_t a1 = u64 & 0xFF; u64 = u64 >> 8; // 0xcd
4 uint8_t a2 = u64 & 0xFF; u64 = u64 >> 8; // 0xab
5 uint8_t a3 = u64 & 0xFF; u64 = u64 >> 8; // 0x89
6 uint8_t a4 = u64 & 0xFF; u64 = u64 >> 8; // 0x67
7 uint8_t a5 = u64 & 0xFF; u64 = u64 >> 8; // 0x45
8 uint8_t a6 = u64 & 0xFF; u64 = u64 >> 8; // 0x23
9 uint8_t a7 = u64 & 0xFF; // 0x01
10 uint64_t r64 = a0; // 0xef
11 r64 = r64 << 8 | a1; // 0xefcd
12 r64 = r64 << 8 | a2; // 0xefcdab
13 r64 = r64 << 8 | a3; // 0xefcdab89
14 r64 = r64 << 8 | a4; // 0xefcdab8967
15 r64 = r64 << 8 | a5; // 0xefcdab896745
16 r64 = r64 << 8 | a6; // 0xefcdab89674523
17 r64 = r64 << 8 | a7; // 0xefcdab8967452301
```

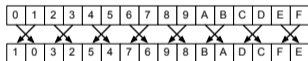
- 29 instructions → SIMD specialized shuffle instructions are very efficient!



# Reverse Lanes Order – vrevq – Full Length

## 1 Shuffle & Permutation

- 3 instructions:
  - vrev16q: to reverse two 8-bit packets
  - vrev32q: to reverse packets of four 8-bit or two 16-bit
  - vrev64q: to reverse packets of eight 8-bit, four 16-bit or two 32-bit
- But **no reverse on the whole register** (no rev128q)



vrev16q\_8



vrev32q\_8



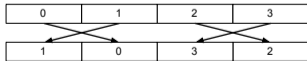
vrev64q\_8



vrev32q\_16



vrev64q\_16



vrev64q\_32

vrevq (128-bit register)

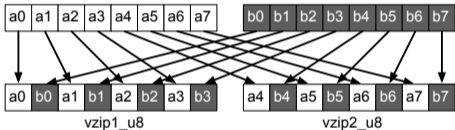


# Interleave – vzip

1 Shuffle & Permutation

- 1:2 interleave of 2 registers

— Example in NEONv1 (with structures)



- Very useful when dealing with **complex numbers** (from SoA to AoS)

```
1 uint8x8_t a = (uint8x8_t) {a0, a1, a2, a3, a4, a5, a6, a7};
2 uint8x8_t b = (uint8x8_t) {b0, b1, b2, b3, b4, b5, b6, b7};
3 uint8x8x2_t s = {a, b}; // struct. of 2 SIMD registers
4 uint8x8x2_t d = vzip_u8(s);
5 uint8x8_t l = d.val[0]; // [a0 b0 a1 b1 a2 b2 a3 b3]
6 uint8x8_t r = d.val[1]; // [a4 b4 a5 b5 a6 b6 a7 b7]
```

— Example in NEONv2 (ARMv8)

```
1 uint8x8_t l = vzip1_u8(a, b); // [a0 b0 a1 b1 a2 b2 a3 b3]
2 uint8x8_t r = vzip2_u8(a, b); // [a4 b4 a5 b5 a6 b6 a7 b7]
```

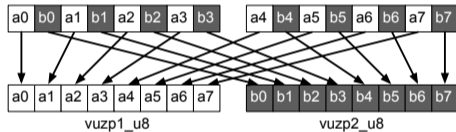


# Deinterleave – vuzp

1 Shuffle & Permutation

- 1:2 deinterleave of 2 registers

— Example in NEONv1 (with structures)



- Very useful when dealing with **complex numbers** (from AoS to SoA)

```
1 uint8x8_t l = (uint8x8_t) {a0, b0, a1, b1, a2, b2, a3, b3};
2 uint8x8_t r = (uint8x8_t) {a4, b4, a5, b5, a6, b6, a7, b7};
3 uint8x8x2_t s = {l, r}; // struct. of 2 SIMD registers
4 uint8x8x2_t d = vuzp_u8(s);
5 uint8x8_t a = d.val[0]; // [a0 a1 a2 a3 a4 a5 a6 a7]
6 uint8x8_t b = d.val[1]; // [b0 b1 b2 b3 b4 b5 b6 b7]
```

— Example in NEONv2 (ARMv8)

```
1 uint8x8_t a = vuzp1_u8(l, r); // [a0 a1 a2 a3 a4 a5 a6 a7]
2 uint8x8_t b = vuzp2_u8(l, r); // [b0 b1 b2 b3 b4 b5 b6 b7]
```



# $4 \times 4$ Transposition

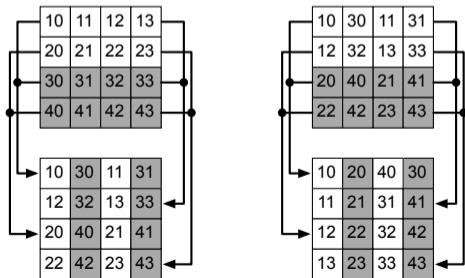
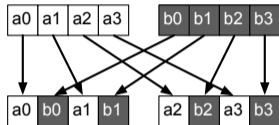
1 Shuffle & Permutation

- 4 registers of 4 elements ( $n = \text{card} = 4$ )

- Squared matrix of  $n \times n$  elements
- $v_0 = [10 \ 11 \ 12 \ 13]$
- $v_1 = [20 \ 21 \ 22 \ 23]$
- $v_2 = [30 \ 31 \ 32 \ 33]$
- $v_3 = [40 \ 41 \ 42 \ 43]$
- Interleave of  $v_i$  with  $v_{n/2+i}$

- Complexity

- $4 \times$  (“vzip1 & vzip2”)
- 8 instructions for 16 elements
- 0.5 instructions per element
- **Very efficient**





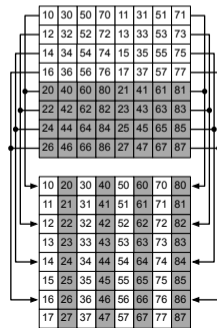
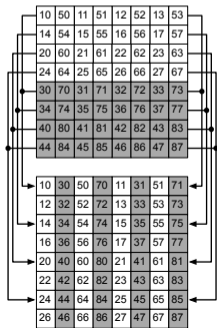
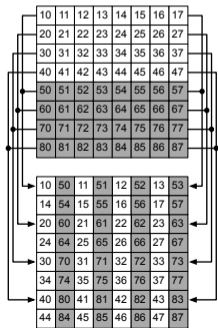
# 8 × 8 Transposition

## 1 Shuffle & Permutation

- 8 registers of 8 elements ( $n = \text{card} = 8$ )
  - Squared matrix of  $n \times n$  elements
  - Interleave of  $v_i$  with  $v_{n/2+i}$

- Complexity

- $12 \times$  (“vzip1 & vzip2”)
- 24 instructions for 64 elements
- 0.375 instructions per element
- **Even more efficient**







# Matrix Transposition – Generalization – Part 1

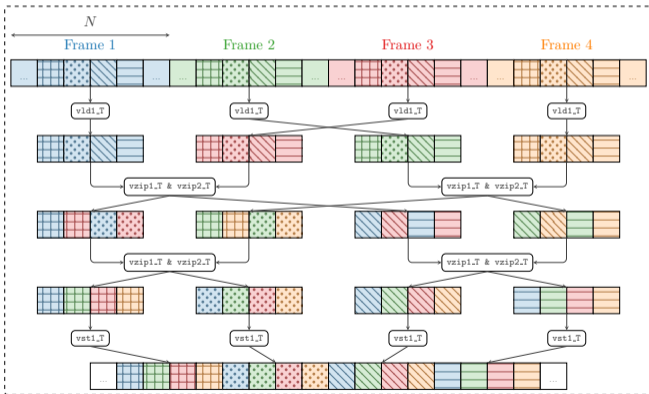
## 1 Shuffle & Permutation

- Logarithmic progression,  $n \times \log_2(n)$  instructions
  - $n = 4$ ,  $n \times \log_2(n) = 4 \times 2 = 8$  instructions,  $n^2 = 16$  elements  
→ 0.5 instructions per elements
  - $n = 8$ ,  $n \times \log_2(n) = 8 \times 3 = 24$  instructions,  $n^2 = 64$  elements  
→ 0.375 instructions per elements
  - $n = 16$ ,  $n \times \log_2(n) = 16 \times 4 = 64$  instructions,  $n^2 = 256$  elements  
→ 0.25 instructions per elements
- Conclusion
  - Transposition is a very efficient operation on SIMD CPUs
  - Even if it requires a lot of instructions (ex.: 64 for  $n = 16$ )



# Matrix Transposition – Generalization – Part 2

## 1 Shuffle & Permutation



- What if we want to transpose a rectangular matrix?
  - Here we consider  $N$ , the number of bits in one frame, as the 1<sup>st</sup> dim of the matrix
  - And  $n$ , the number of lanes in the SIMD registers (and the number of frames), as the 2<sup>nd</sup> dimension of the matrix
- $n \times \log_2(n) \times \frac{N}{n}$  `vzipx_T` instructions
  - $n \times \log_2(n) \rightarrow$  “horizontal computations”
  - Repeated  $\frac{N}{n}$  times  $\rightarrow$  “vertical computations”



# Memory Deinterleave – vld2 / vld3 / vld4

1 Shuffle & Permutation

- For SIMD registers: `vzip` et `vuzp` for 1:2 interleave
- Load from memory: `vld2` / `vld2` / `vld4` for 1:2, 1:3 and 1:4 deinterleave
- Store into memory: `vst2` / `vst3` / `vst4` for 1:2, 1:3 and 1:4 interleave
- New types (= structures containing multiple registers)
  - 3 pseudo-structures: `x2_t`, `x3_t` and `x4_t` (as type suffix)
  - Access to structure fields: `.val[0]`, `.val[1]`, `.val[2]` and `.val[3]`



# Memory Deinterleave – vld2 / vld3 / vld4

## 1 Shuffle & Permutation

- Example # 1: Complex numbers (1:2)

```
1 float32_t Z[8] = {0, 1, 2, 3, 4, 5, 6, 7};
2 float32x4x2_t z = vld2q_f32(&Z[0]); // load with 1:2 deinterleaving
3 float32x4_t r = z.val[0];           // r = [0 2 4 6], register #0 acces
4 float32x4_t c = z.val[1];           // c = [1 3 5 7], register #1 acces
5
6 z.val[0] = c; z.val[1] = r;         // permutation
7 vst2q_f32(&Z[0], z);               // Z = [1 0 3 2 5 4 7 6], store with 1:2 interleaving
```

- Example # 2: 32-bit RGBA color images (1:4)

```
1 uint32_t C[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F};
2 uint32x4x4_t c = vld4q_f32(&C[0]); // load with 1:4 deinterleaving
3 uint32x4_t r = z.val[0];           // r = [0 4 8 C], register #0 acces
4 uint32x4_t g = z.val[1];           // g = [1 5 9 D], register #1 acces
5 uint32x4_t b = z.val[2];           // b = [2 6 A E], register #2 acces
6 uint32x4_t a = z.val[3];           // a = [3 7 B F], register #3 acces
```



# Crossbar Shuffles – vtbl1

## 1 Shuffle & Permutation

- 1-input cross-bar
  - NEONv2 comes with a 1-input HW shuffle unit →  $d = \text{vtbl}(a, c): d_i = a[c_i]$
  - Register-indexed addressing with  $c_i \in [0, \text{card} - 1]$
  - To perform **all** possible shuffles
- Reproduce existing regular shuffles

```
1 uint8x8_t a = (uint8x8_t)    {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t c = (uint8x8_t)    { 1,  0,  3,  2,  5,  4,  7,  6};
3 uint8x8_t d = vtbl1_u8(a,c); // [11 10 13 12 15 14 17 16] <- same behavior as with 'vrev16_u8'
4         c = (uint8x8_t)    { 3,  2,  1,  0,  7,  6,  5,  4};
5         d = vtbl1_u8(a,c); // [13 12 11 10 17 16 15 14] <- same behavior as with 'vrev32_u8'
```

- And all the irregular shuffles

```
1 uint8x8_t a = (uint8x8_t)    {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t c = (uint8x8_t)    { 1,  3,  3,  7,  7,  7,  5,  5};
3 uint8x8_t d = vtbl1_u8(a,c); // [11 13 13 17 17 17 15 15]
```



# Crossbar Shuffles – vtbl1

## 1 Shuffle & Permutation

- 1-input cross-bar
  - NEONv2 comes with a 1-input HW shuffle unit  $\rightarrow d = \text{vtbl}(a, c) = a[c_i]$
  - Register-indexed addressing with  $c_i \in [0, \text{card} - 1]$
  - To perform **all** possible shuffles
- Reproduce existing regular shuffles

```
1 uint8x8_t a = (uint8x8_t)    {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t c = (uint8x8_t)    { 1,  0,  3,  2,  5,  4,  7,  6};
3 uint8x8_t d = vtbl1_u8(a,c); // [11 14 13 12 15 14 17 16] <- same behavior as with 'vrev16_u8'
4         c = (uint8x8_t)    { 1,  0,  7,  6,  5,  4};
5         d = vtbl1_u8(a,c); // [12 11 10 17 16 15 14] <- same behavior as with 'vrev32_u8'
```

- And all the **regular** shuffles

```
1 uint8x8_t a = (uint8x8_t)    {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t c = (uint8x8_t)    { 1,  3,  3,  7,  7,  7,  5,  5};
3 uint8x8_t d = vtbl1_u8(a,c); // [11 13 13 17 17 17 15 15]
```

Regular shuffles are more efficient!



# Crossbar Shuffles – vtbl2 / vtbl3 / vtbl4

## 1 Shuffle & Permutation

- 2/3/4-input cross-bar
  - Aggregation of registers to form a super register with continuous indexing
  - $c_i \in [0, 2 \times \text{card} - 1], [0, 3 \times \text{card} - 1], [0, 4 \times \text{card} - 1]$
- Code example (1:2, 1:3 and 1:4)

```
1 uint8x8_t  v0 = (uint8x8_t)      {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t  v1 = (uint8x8_t)      {18, 19, 20, 21, 22, 23, 24, 25};
3 uint8x8_t  v2 = (uint8x8_t)      {26, 27, 28, 29, 30, 31, 32, 33};
4 uint8x8_t  v3 = (uint8x8_t)      {34, 35, 36, 37, 38, 39, 40, 41};
5 uint8x8x2_t s2 = {v0, v1};
6 uint8x8x3_t s3 = {v0, v1, v2};
7 uint8x8x4_t s4 = {v0, v1, v2, v4};
8 uint8x8_t  c2 = (uint8x8_t)      { 0,  2,  4,  6,  8, 10, 12, 14}; // 1:2
9 uint8x8_t  c3 = (uint8x8_t)      { 0,  3,  6,  9, 12, 15, 18, 21}; // 1:3
10 uint8x8_t  c4 = (uint8x8_t)      { 0,  4,  8, 12, 16, 20, 24, 28}; // 1:4
11 uint8x8_t  y2 = vtbl2_u8(s2, c2); // [10 12 14 16 18 20 22 24]
12 uint8x8_t  y3 = vtbl3_u8(s3, c3); // [10 13 16 19 22 25 28 31]
13 uint8x8_t  y4 = vtbl4_u8(s4, c4); // [10 14 18 22 26 30 34 38]
```



# Crossbar Shuffles – vtb12 / vtb13 / vtb14

## 1 Shuffle & Permutation

- 2/3/4-input cross-bar
  - Aggregation of registers to form a super register with continuous indexing
  - $c_i \in [0, 2 \times \text{card} - 1], [0, 3 \times \text{card} - 1], [0, 4 \times \text{card} - 1]$
- Code example (1:2, 1:3 and 1:4)

```
1 uint8x8_t v0 = (uint8x8_t) {10, 11, 12, 13, 14, 15, 16, 17};
2 uint8x8_t v1 = (uint8x8_t) {18, 19, 20, 21, 22, 23, 24, 25};
3 uint8x8_t v2 = (uint8x8_t) {26, 27, 28, 29, 30, 31, 32, 33};
4 uint8x8_t v3 = (uint8x8_t) {34, 35, 36, 37, 38, 39, 40, 41};
5 uint8x8x2_t s2 = {v0, v1};
6 uint8x8x3_t s3 = {v0, v1, v2};
7 uint8x8x4_t s4 = {v0, v1, v2, v3};
8 uint8x8_t c2 = (uint8x8_t) { 0, 2, 4, 6, 8, 10, 12, 14}; // 1:2
9 uint8x8_t c3 = (uint8x8_t) { 0, 3, 6, 9, 12, 15, 18, 21}; // 1:3
10 uint8x8_t c4 = (uint8x8_t) { 0, 4, 8, 12, 16, 20, 24, 28}; // 1:4
11 uint8x8_t y2 = vtb12_u8(s2, c2); // [10 12 14 16 18 20 22 24]
12 uint8x8_t y3 = vtb13_u8(s3, c3); // [10 13 16 19 22 25 28 31]
13 uint8x8_t y4 = vtb14_u8(s4, c4); // [10 14 18 22 26 30 34 38]
```

These shuffles take more than 1 CPU cycle!





# Table of Contents

2 Reduction

▶ Shuffle & Permutation

▶ Reduction

▶ Fixed-Point

▶ Mixed-Precision



# Introduction

## 2 Reduction

- Reduction operations involve the elements inside the same registers
  - Reduction operation = horizontal operation
  - That's not why SIMD instructions were intended for at the beginning!
  - But sometime this is mandatory in the code... (ex.: **map-reduce** pattern)
- Example of a reduction (addition)

```
1 uint8x8_t a = (uint8x8_t) { 1, 2, 3, 4, 5, 6, 7, 8 };  
2 uint8_t r   = v???_u8(a,b); // r = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36
```

- **How to do this in SIMD?**



# General Case – Reduction Tree

## 2 Reduction

- Let us suppose we don't have specialized SIMD instructions
- Example of an 8-bit integers horizontal addition (= sum)

```
1 uint8x16_t a = (uint8x16_t)          { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
2 uint8x8_t l = vget_low_u8(a);
3 uint8x8_t h = vget_high_u8(a);      // a = [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
4 uint8x16_t r128 = vcombine_u8(h, l); // r128 = [ 8  9 10 11 12 13 14 15 0  1  2  3  4  5  6  7]
5 uint8x16_t sum = vaddq_u8(a, r128);  // sum = [ 8 10 12 14 16 18 20 22 8 10 12 14 16 18 20 22]
6 uint8x16_t r64 = vrev64q_u32(sum);   // r64 = [ 16 18 20 22 8 10 12 14 16 18 20 22 8 10 12 14]
7 sum = vaddq_u8(sum, r64);           // sum = [ 24 28 32 36 24 28 32 36 24 28 32 36 24 28 32 36]
8 uint8x16_t r32 = vrev32q_u16(sum);   // r32 = [ 32 36 24 28 32 36 24 28 32 36 24 28 32 36 24 28]
9 sum = vaddq_u8(sum, r32);           // sum = [ 56 64 56 64 56 64 56 64 56 64 56 64 56 64 56 64]
10 uint8x16_t r16 = vrev16q_u8(sum);   // r16 = [ 64 56 64 56 64 56 64 56 64 56 64 56 64 56 64 56]
11 sum = vaddq_u8(sum, r16);           // sum = [120 120 120 120 120 120 120 120 120 120 120 120 120 120 120]
12 uint8_t ssum = vgetq_lane_u8(sum, 0); // from SIMD register 'sum' to scalar register 'ssum'
```

- Number of instructions
  - For 16 elements: 4 regular shuffles (vcombine, vrev64q, vrev32q and vrev16q)
  - For 16 elements: 3 vertical additions (vaddq)
  - In general:  $2 \times \log_2(n - 1) + 1$  instructions



# Horizontal Pairwise Operations

## 2 Reduction

- Pairwise operations = SIMD/SIMD partial reduction
  - For associative operators: OP in { add, min, max }
  - `vpad`, `vpm`, `vpm`
  - `T=vpOP(T,T)` for 64-bit registers
  - `T=vpOPq(T,T)` for 128-bit registers

```
1 uint8x8_t a = (uint8x8_t)    { 1,  2,  3,  4,  5,  6,  7,  8 };
2 uint8x8_t b = (uint8x8_t)    { 9, 10, 11, 12, 13, 14, 15, 16 };
3 uint8x8_t c = vpad_u8(a,b); // [ 3  7 11 15 19 23 27 31 ]
```

- We can use pairwise operation at the last level of the previous reduction tree  
→ Replaces 1 shuffle (`vrev16`) and 1 addition (`vadd`)!
- There are other pairwise operations in NEON...



# Across Horizontal Pairwise Operations

## 2 Reduction

- Operation *across vector* = full reduction SIMD to scalar value
  - $T = \text{vaOP}(TxM\_t)$ , OP in { add, min, max }

```
1 uint8x8_t c = (uint8x8_t) { 1, 2, 3, 4, 5, 6, 7, 8 };
2 uint8_t   r = vaddv_u8(c); // r = 36
```

- Operation *across long vector*
  - $TT = \text{vaddlv}(TxM\_t)$  for accumulation without *overflow*

```
1 uint8x8_t c = v_mov_n_u8(255); // c = {255, 255, 255, 255, 255, 255, 255, 255}
2 uint8_t   r8 = vaddv_u8(c);    // r8 = 248 // overflow
3 uint16_t  r16 = vaddlv_u8(c);  // r16 = 2048 // 8*255
```

- What for?
  - For while-loops and do-while-loops
  - To exit as soon as an element in the SIMD register is true (or false)
  - To continue until all elements of a SIMD register are true



# Table of Contents

3 Fixed-Point

- ▶ Shuffle & Permutation
- ▶ Reduction
- ▶ Fixed-Point
- ▶ Mixed-Precision



## Quantized Conversion – `vcvt_n`

### 3 Fixed-Point

- For fixed-point computations
  - Compute the coefficients of a floating-point filter
  - Fixed-point computations for greater parallelism
- Conversion with  $2^p$  quantization from `float` to `int` and `uint`
  - `s = vcvt_n_s_f (f, p)`, `u = vcvt_n_u_f(f, p)` (with rounding to zero)
- Conversion with  $1/2^p$  quantization from `int` and `uint` to `float`
  - `f = vcvt_n_f_s(s, p)`, `f = vcvt_n_f_u(u, p)` (with rounding to zero)
- Available for all 3 registers types and 2 sizes:
  - $\{ f16, f32, f64 \} \times \{ 64\text{-bit}, 128\text{-bit} \}$  (to signed and unsigned integers)

```
1 float32x4_t f = (float32x4_t)          { -1.75f, -1.25f, 1.25f, 1.75f };
2 int32x4_t   i = vcvtq_n_s32_f32(f, 10); // [ -1792 -1280 1280 1792 ]
3           f = vcvtq_n_f32_s32(i, 10);  // [ -1.75f -1.25f 1.25f 1.75f ]
```



# Table of Contents

4 Mixed-Precision

▶ Shuffle & Permutation

▶ Reduction

▶ Fixed-Point

▶ Mixed-Precision





# Float to Float Conversion – `vcvt_f`

4 Mixed-Precision

- One of the few things incomplete...

```
1 float16x4_t vcvt_f16_f32(float32x4_t a);
2 float32x2_t vcvt_f32_f64(float64x2_t a);
3 float32x4_t vcvt_f32_f16(float16x4_t a);
4 float64x2_t vcvt_f64_f32(float32x2_t a);
5
6 float16x8_t vcvt_high_f16_f32(float16x4_t r, float32x4_t a);
7 float32x4_t vcvt_high_f32_f64(float32x2_t r, float64x2_t a);
8 float32x4_t vcvt_high_f32_f16(float16x8_t a);
9 float64x2_t vcvt_high_f64_f32(float32x4_t a);
10
11 float32x2_t vcvtx_f32_f64(float64x2_t a);
12 float32x4_t vcvtx_high_f32_f64(float32x2_t r, float64x2_t a);
```

- Missing `vcvt_low` instructions



# Compute in 32-bit float and Store in 16-bit float

4 Mixed-Precision

- This is possible thanks to the previous `vcvt_f` instructions
- Depending on the architecture it can be costly to convert 32-bit floats to 16-bit floats (and 16-bit to 32-bit): **this is not free!**
- Useful when the limitation comes from the **memory bandwidth**
- Compute in 32-bit and store in 16-bit is more accurate than compute and store in 16-bit
  - Still there is a loss of precision and accuracy when converting 32-bit float to 16-bit float



# From Smaller to Bigger Integers (Promotion)

## 4 Mixed-Precision

- Convert one `uint8_t` to two `uint16_t`
  1. Separate low and high parts of the `uint8_t` register (`vget_low` and `vget_high`)
  2. Extend both parts from 8-bit to 16-bit representation (`vmovl_u8`)
- Convert one `uint16_t` to two `uint32_t`
  1. Separate low and high parts of the `uint16_t` register (`vget_low` and `vget_high`)
  2. Extend both parts from 16-bit to 32-bit representation (`vmovl_u16`)

```
1 // conversion (= promotion) from uint8 to uint16
2 uint8x16_t v = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F};
3 uint16x8_t v_lo = vmovl_u8(vget_low_u8(v)); // v_lo = [0 1 2 3 4 5 6 7]
4 uint16x8_t v_hi = vmovl_u8(vget_high_u8(v)); // v_hi = [8 9 A B C D E F]
5 // conversion (= promotion) from uint16 to uint32
6 uint32x4_t v_lo_lo = vmovl_u16(vget_low_u16(v_lo)); // v_lo_lo = [0 1 2 3]
7 uint32x4_t v_lo_hi = vmovl_u16(vget_high_u16(v_lo)); // v_lo_hi = [4 5 6 7]
8 uint32x4_t v_hi_lo = vmovl_u16(vget_low_u16(v_hi)); // v_hi_lo = [8 9 A B]
9 uint32x4_t v_hi_hi = vmovl_u16(vget_high_u16(v_hi)); // v_hi_hi = [C D E F]
```



# From Bigger to Smaller Integers

## 4 Mixed-Precision

- Convert two `uint16_t` to one `uint8_t`
  1. Truncate both 16-bit registers (`vqmovn_u16`)  $\rightarrow$  `uint8x8_t`
  2. Combine two `uint8x8_t` registers into a full `uint8x16_t` register (`vcombine_u8`)
- Convert two `uint32_t` to one `uint16_t`
  1. Truncate both 32-bit registers (`vqmovn_u32`)  $\rightarrow$  `uint16x4_t`
  2. Combine two `uint16x4_t` registers into a full `uint16x8_t` register (`vcombine_u16`)

```
1 uint32x4_t v_lo_lo = {0, 1, 2, 3};
2 uint32x4_t v_lo_hi = {4, 5, 6, 7};
3 uint32x4_t v_hi_lo = {8, 9, A, B};
4 uint32x4_t v_hi_hi = {C, D, E, F};
5 // conversion from uint32 to uint16
6 uint16x8_t v_lo = vcombine_u16(vqmovn_u32(v_lo_lo), vqmovn_u32(v_lo_hi)); // v_lo = [0 1 2 3 4 5 6 7]
7 uint16x8_t v_hi = vcombine_u16(vqmovn_u32(v_hi_lo), vqmovn_u32(v_hi_hi)); // v_hi = [8 9 A B C D E F]
8 // conversion from uint16 to uint8
9 uint8x16_t v = vcombine_u8(vqmovn_u16(v_lo), vqmovn_u16(v_hi)); // [0 1 2 3 4 5 6 7 8 9 A B C D E F]
```



## Q&A

*Thank you for listening!*  
*Do you have any questions?*