CHAOSGROUP

стаж

6 стажантски позиции, **3** месечен платен стаж, **4** часов работен ден.

V-Ray Core:

- C++ Developer
- C++ Developer
- C++ Developer
- C++ & Haskell Developer

V-Ray Cloud:

- C++ Developer
- JavaScript Developer

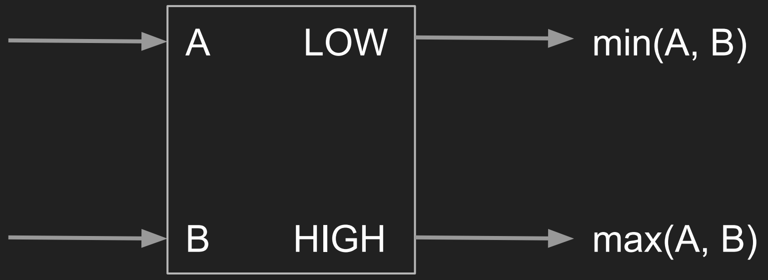
Sorting Networks

High Performance Computing - FMI, Fall 2015

martin.krastev@chaosgroup.com

Who invented them?

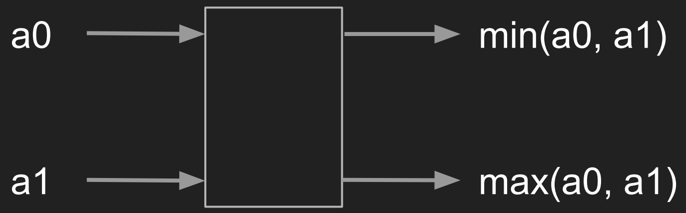
- 1954 - groundwork by Armstrong, Nelson and O'Connor (see Knuth)
- 1968 - advanced by Ken Batcher - one of the pioneers of massive parallelism
    - Fundamental building block -- the **comparison function**



    - Sorts an array in ½ * p * (p+1) steps, p = log2(N), via **parallelism**, so O(log(N) * log(N)), always

Sorting it all out

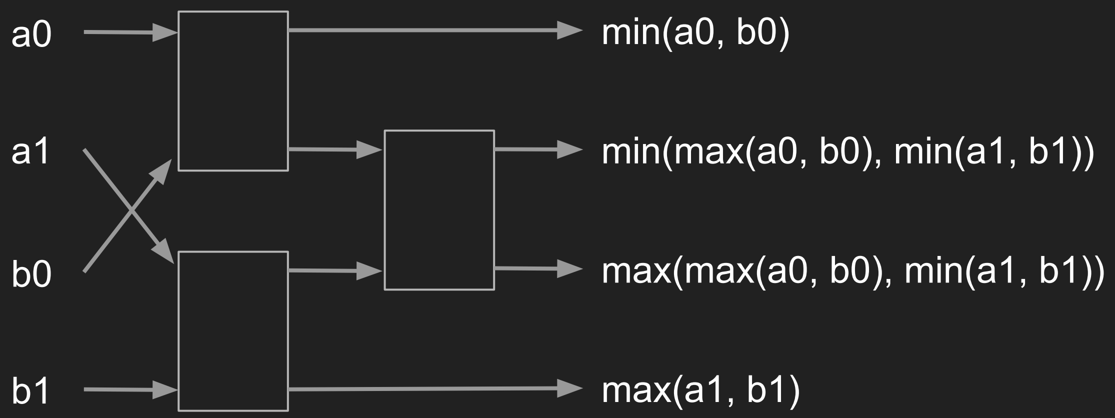
Sorting an array of **two** elements in ascening order:



Hurray?

Sorting it all out, cont'd

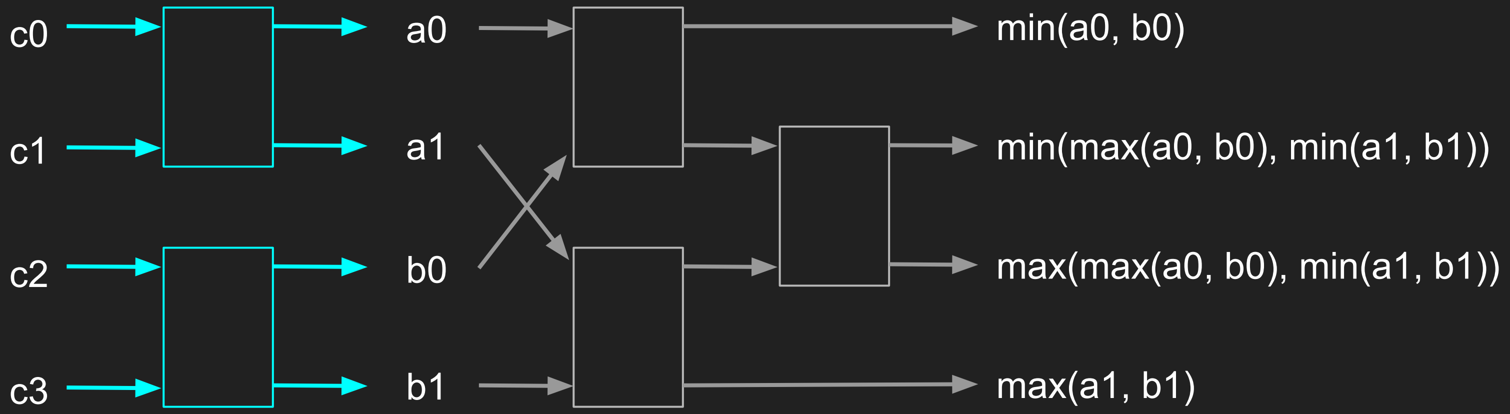
Merging **two** sorted arrays of **two** elements each, producing a third sorted array:



..Yay?

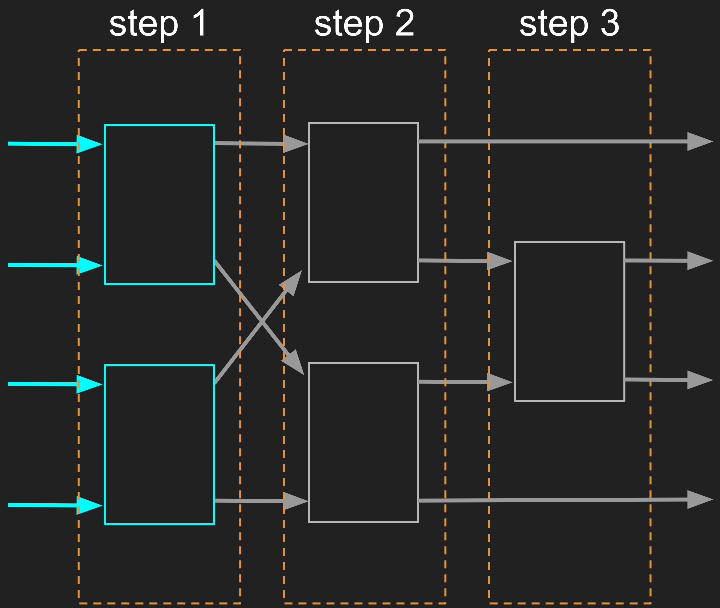
Sorting it all out, cont'd

Sorting an array of **four** elements:



Sorting it all out, cont'd

..Via **parallelism**:



p = log2(4) = 2, steps = ½ * p * (p + 1) = ½ * 2 * 3 = **3**

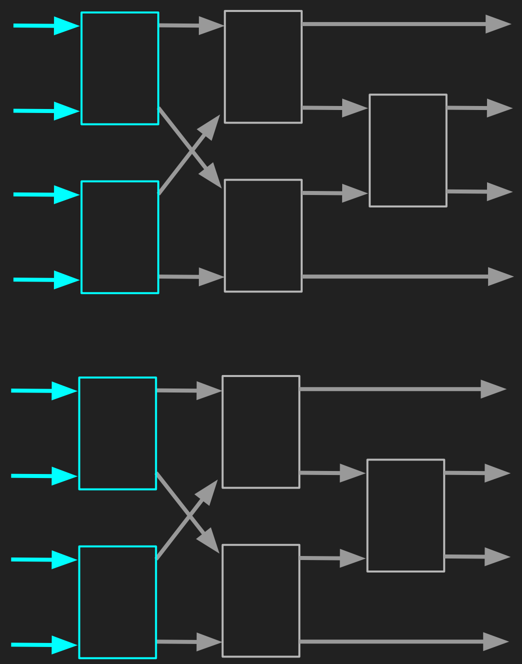
‘Bout that homework..

```
static void foo( // Ken Batcher's odd-even sorting network
   float (& inout)[8]) {

   const size_t idx[][2] = {
         { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 },
         { 0, 2 }, { 1, 3 }, { 4, 6 }, { 5, 7 },
         { 1, 2 }, { 5, 6 },
         { 0, 4 }, { 1, 5 }, { 2, 6 }, { 3, 7 },
         { 2, 4 }, { 3, 5 },
         { 1, 2 }, { 3, 4 }, { 5, 6 }
   };

   for (size_t i = 0; i < sizeof(idx) / sizeof(idx[0]); ++i) {
         const float x = inout[idx[i][0]];
         const float y = inout[idx[i][1]];

         inout[idx[i][0]] = std::min(x, y);
         inout[idx[i][1]] = std::max(x, y);
   }
}
```

$p = \log2(8) = 3$, $\text{steps} = \frac{1}{2} * p * (p + 1) = \frac{1}{2} * 3 * 4 = \mathbf{6}$

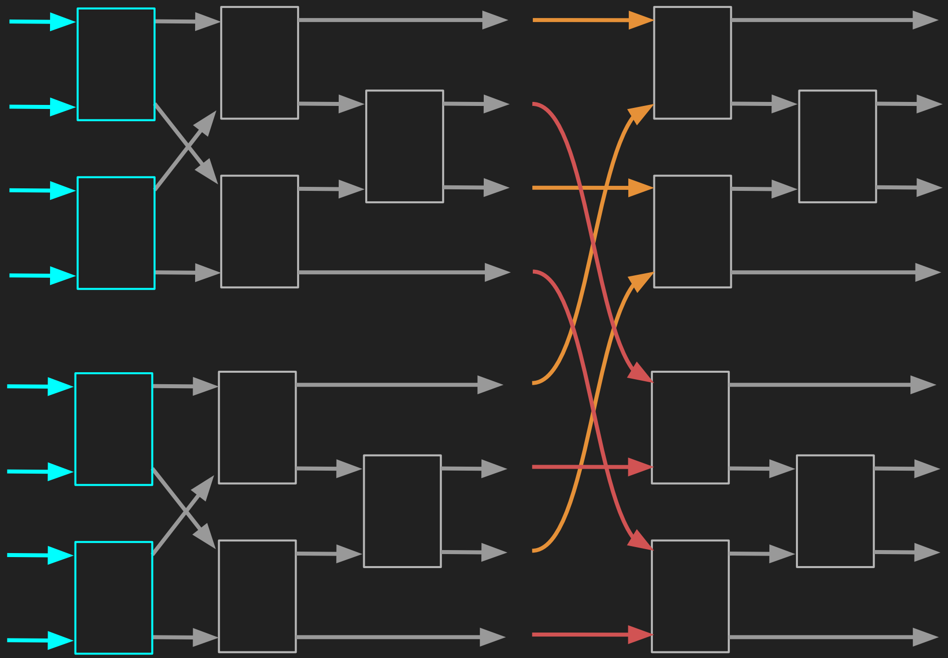
‘Bout that homework, cont’d

First three rows of idx sort the two **four**-element halves of input independently..



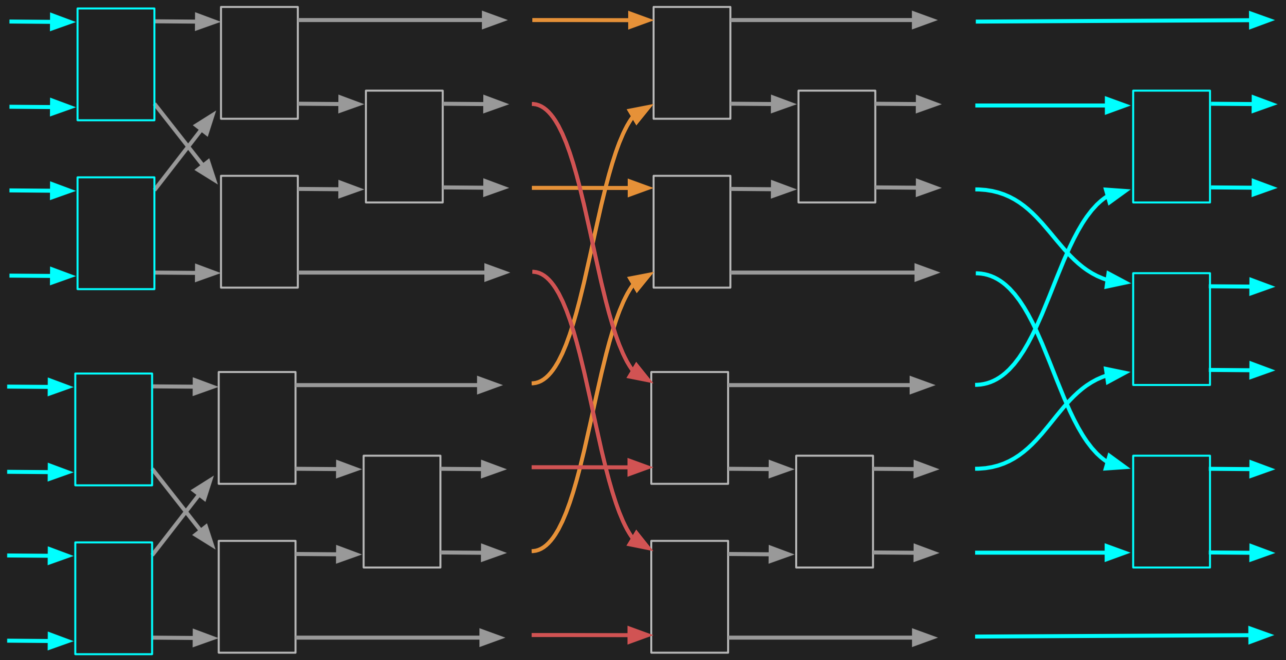
‘Bout that homework, cont’d

Rows 4 and 5 merge the two sorted halves, **odd** with **odd**, **even** with **even**..



‘Bout that homework, cont’d

Last row concludes the interior comparisons, merging odd and even rows



Just look at all this parallelism!

Isn’t it beautiful?

- We started with 4 sortings of 2-element arrays..
- Continued with 2 sortings of 4-element arrays..
- Then merged the two in an odd-only and an even-only sorts..
- And finally combined the odd and even lanes in the interior

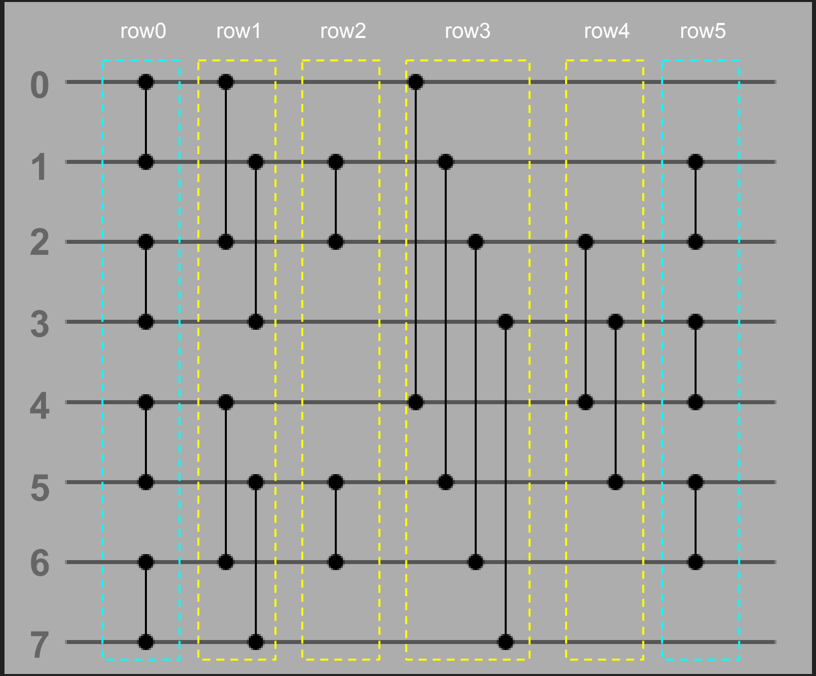
Who said SIMD?

All this parallelism is begging for a SIMD implementation. You were supposed to exploit that in homework #2

- A few of you did!
- The remainder of this talk is dedicated to those of you who didn't :)
- We will focus on Batcher's Odd-Even network, leaving the Bitonic network to your curiosity.

So, let’s SIMD-ify!

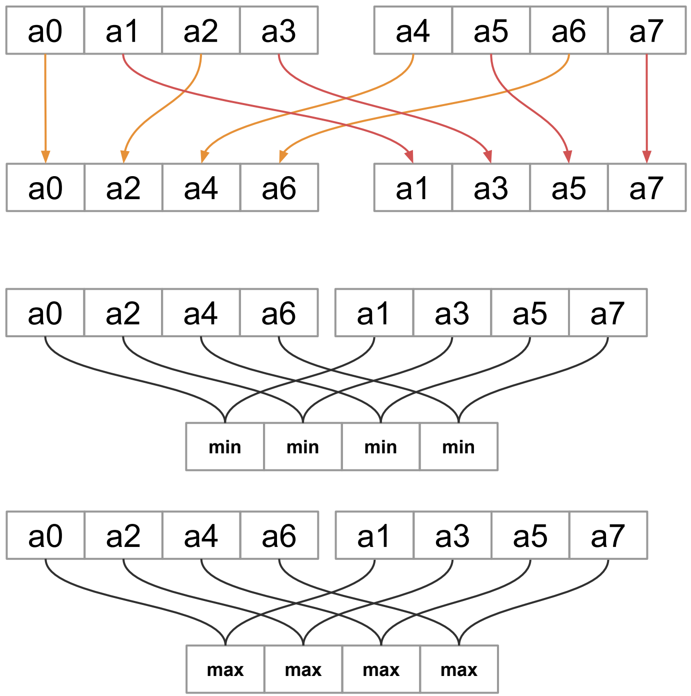
But first, a graphic notation* of the idx array and the comparisons it encodes:



* Courtesy of [Wikipedia]

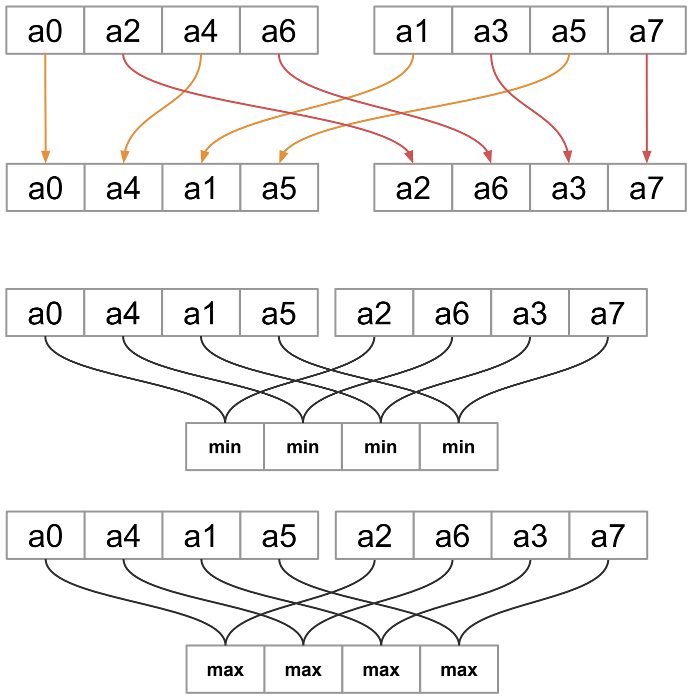
Odd-even in SIMD (SSE4.1)

```
static void foo( // odd_even_simd_sort
    float (& inout)[8]) {

    const __m128 r0_in0 = _mm_load_ps(inout + 0); // 0, 1, 2, 3
    const __m128 r0_in1 = _mm_load_ps(inout + 4); // 4, 5, 6, 7

    // stage 0
    const __m128 r0_A = _mm_shuffle_ps(r0_in0, r0_in1, 0x88); // 0, 2, 4, 6
    const __m128 r0_B = _mm_shuffle_ps(r0_in0, r0_in1, 0xdd); // 1, 3, 5, 7

    const __m128 r0_min = _mm_min_ps(r0_A, r0_B); // 0, 2, 4, 6
    const __m128 r0_max = _mm_max_ps(r0_A, r0_B); // 1, 3, 5, 7
```



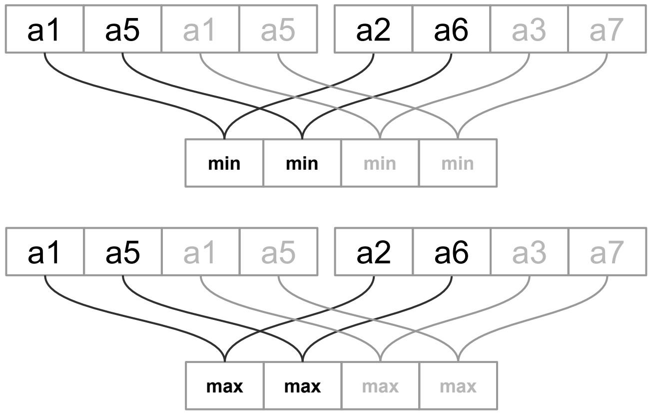
Odd-even in SIMD (SSE4.1), cont'd

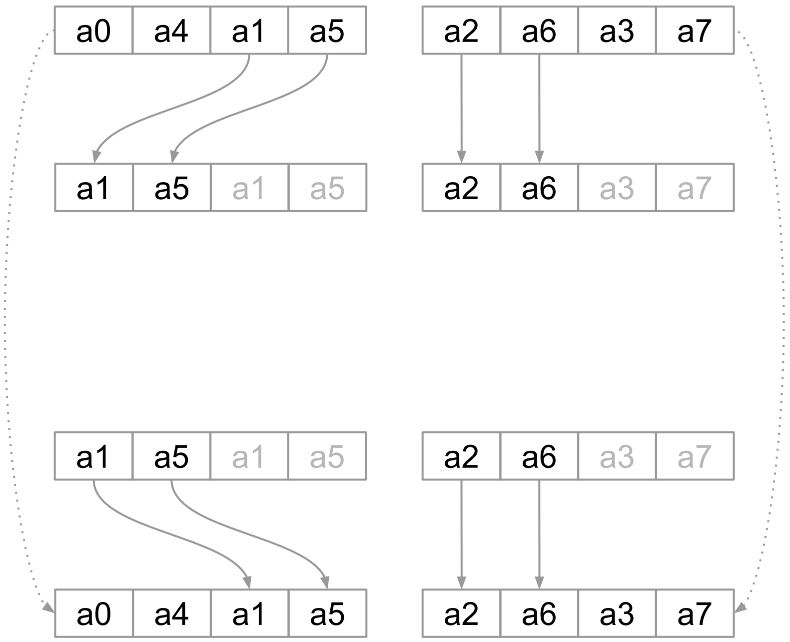
```
// stage 1
const __m128 r1_A = _mm_shuffle_ps(r0_min, r0_max, 0x88); // 0, 4, 1, 5
const __m128 r1_B = _mm_shuffle_ps(r0_min, r0_max, 0xdd); // 2, 6, 3, 7

const __m128 r1_min = _mm_min_ps(r1_A, r1_B); // 0, 4, 1, 5
const __m128 r1_max = _mm_max_ps(r1_A, r1_B); // 2, 6, 3, 7
```



Odd-even in SIMD (SSE4.1), cont'd

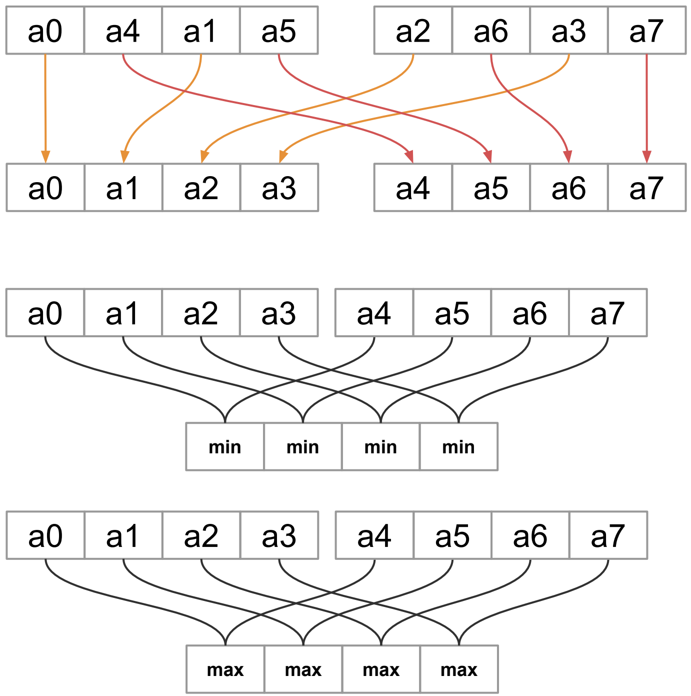
```
// stage 2
const __m128 r2_A = _mm_movehl_ps(r1_min, r1_min); // 1, 5, -, -
const __m128 r2_B = r1_max;                         // 2, 6, -, -

const __m128 r2_min = _mm_min_ps(r2_A, r2_B); // 1, 5, -, -
const __m128 r2_max = _mm_max_ps(r2_A, r2_B); // 2, 6, -, -

const __m128 r2_out0 = _mm_movelh_ps(r1_min, r2_min); // 0, 4, 1, 5
const __m128 r2_out1 = _nn_movell_ps(r1_max, r2_max); // 2, 6, 3, 7
```





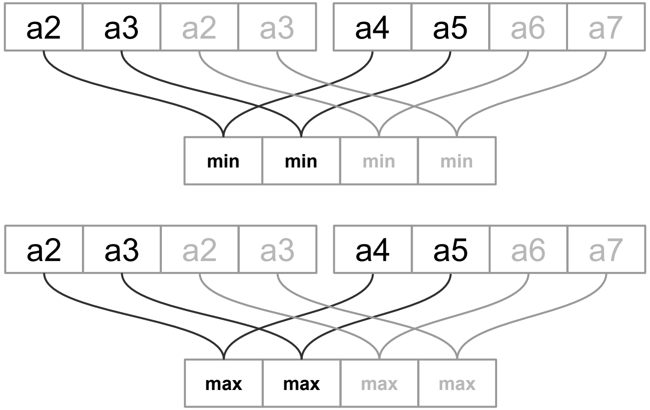
Odd-even in SIMD (SSE4.1), cont'd

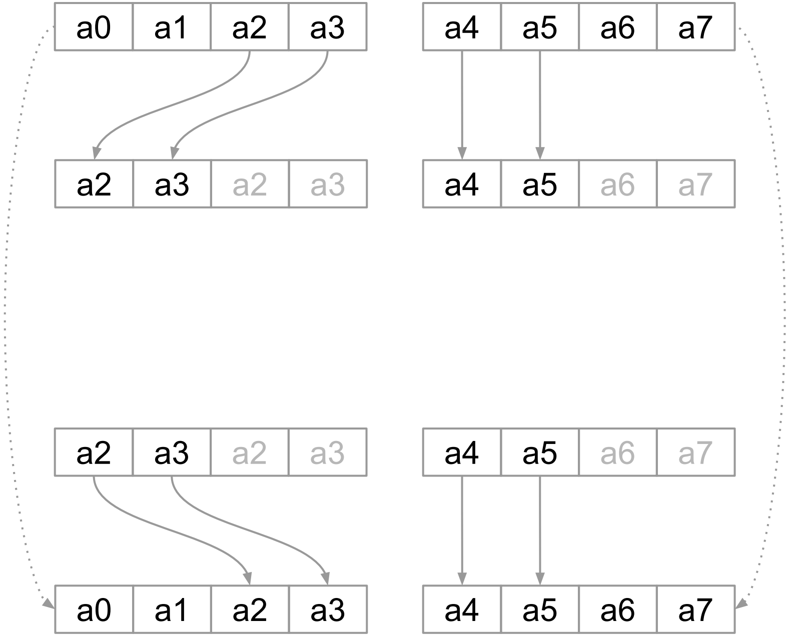
```
// stage 3
const __m128 r3_A = _mm_shuffle_ps(r2_out0, r2_out1, 0x88); // 0, 1, 2, 3
const __m128 r3_B = _mm_shuffle_ps(r2_out0, r2_out1, 0xdd); // 4, 5, 6, 7

const __m128 r3_min = _mm_min_ps(r3_A, r3_B); // 0, 1, 2, 3
const __m128 r3_max = _mm_max_ps(r3_A, r3_B); // 4, 5, 6, 7
```



Odd-even in SIMD (SSE4.1), cont'd

```
// stage 4
const __m128 r4_A = _mm_movehl_ps(r3_min, r3_min); // 2, 3, -, -
const __m128 r4_B = r3_max;                         // 4, 5, -, -

const __m128 r4_min = _mm_min_ps(r4_A, r4_B); // 2, 3, -, -
const __m128 r4_max = _mm_max_ps(r4_A, r4_B); // 4, 5, -, -

const __m128 r4_out0 = _mm_movelh_ps(r3_min, r4_min); // 0, 1, 2, 3
const __m128 r4_out1 = _nn_movell_ps(r3_max, r4_max); // 4, 5, 6, 7
```





Odd-even in SIMD (SSE4.1), cont'd

Five stages so far and the code is simple and efficient. Enter the final stage..

And here things in our Odd-Even network get ugly..

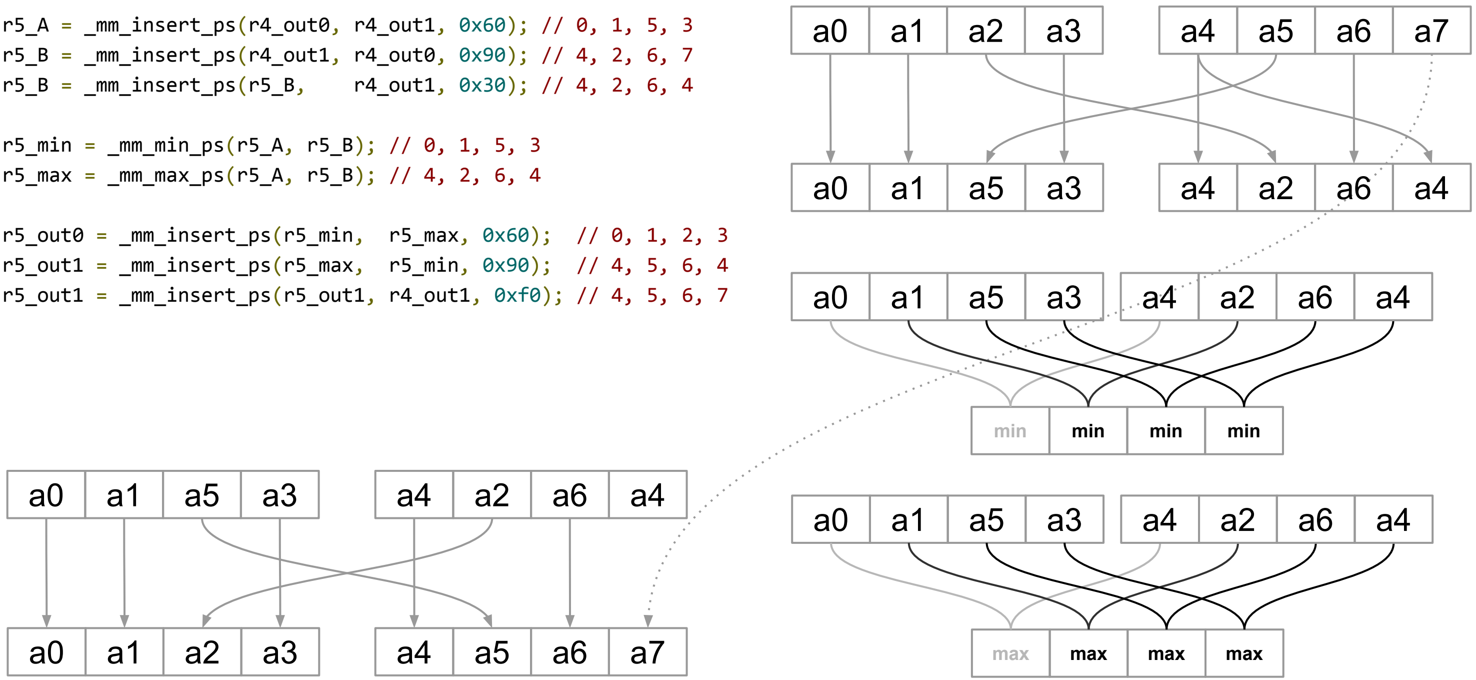
Can you guess why things get ugly?

note: This stage is the reason why

1. We need SSE4.1 for this routine (it looks and performs worse in SSE2)
2. Bitonic networks are more popular (and usually faster on x86-64)

Odd-even in SIMD (SSE4.1), cont'd

```
// stage 5
const __m128 r5_A = _mm_insert_ps(r4_out0, r4_out1, 0x60); // 0, 1, 5, 3
      __m128 r5_B = _mm_insert_ps(r4_out1, r4_out0, 0x90); // 4, 2, 6, 7
             r5_B = _mm_insert_ps(r5_B,    r4_out1, 0x30); // 4, 2, 6, 4

const __m128 r5_min = _mm_min_ps(r5_A, r5_B); // 0, 1, 5, 3
const __m128 r5_max = _mm_max_ps(r5_A, r5_B); // 4, 2, 6, 4

const __m128 r5_out0 = _mm_insert_ps(r5_min,  r5_max, 0x60);  // 0, 1, 2, 3
      __m128 r5_out1 = _mm_insert_ps(r5_max,  r5_min, 0x90);  // 4, 5, 6, 4
             r5_out1 = _mm_insert_ps(r5_out1, r4_out1, 0xf0); // 4, 5, 6, 7
```



Odd-even in SIMD (SSE4.1), cont'd

```
    // output
    _mm_store_ps(inout + 0, r5_out0);
    _mm_store_ps(inout + 4, r5_out1);
}
```

Bonus: routine has a redundant permutation - find it!

References

[1] Ken Batcher, 1968, "Sorting networks and their applications"
[2] Intel Intrinsics Guide