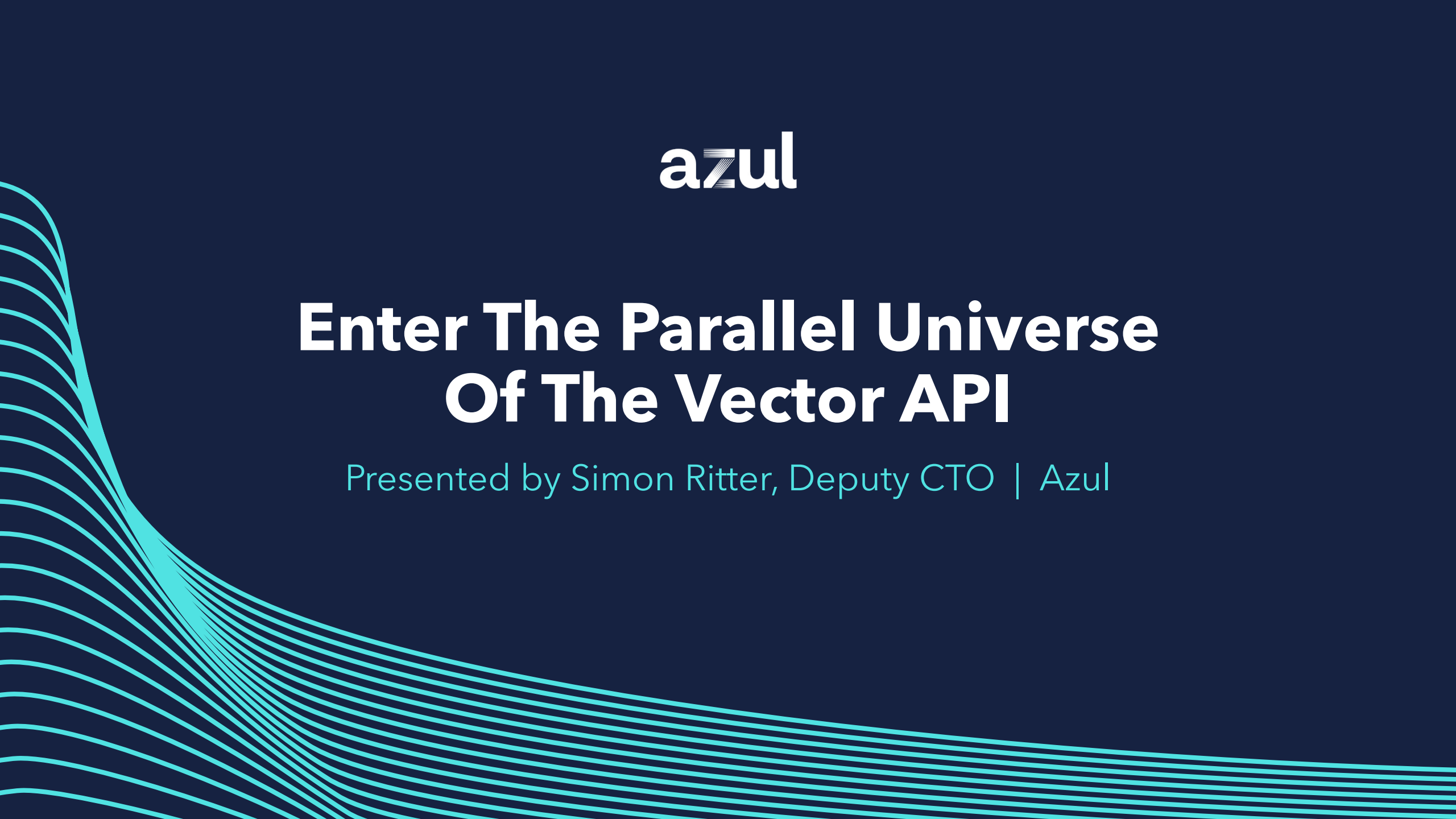


Enter The Parallel Universe Of The Vector API

Presented by Simon Ritter, Deputy CTO | Azul

The Vector API (JDK 15 and Earlier)

```
import java.util.Vector;

Vector<String> myStrings = new Vector<>();

myStrings.add("Foo");

int count = myStrings.size();
```

- This is not what we are here to talk about today

azul

Concurrent or Parallel: What Is The Difference?

Concurrency

Two or more tasks that start, run, and complete in overlapping time periods.

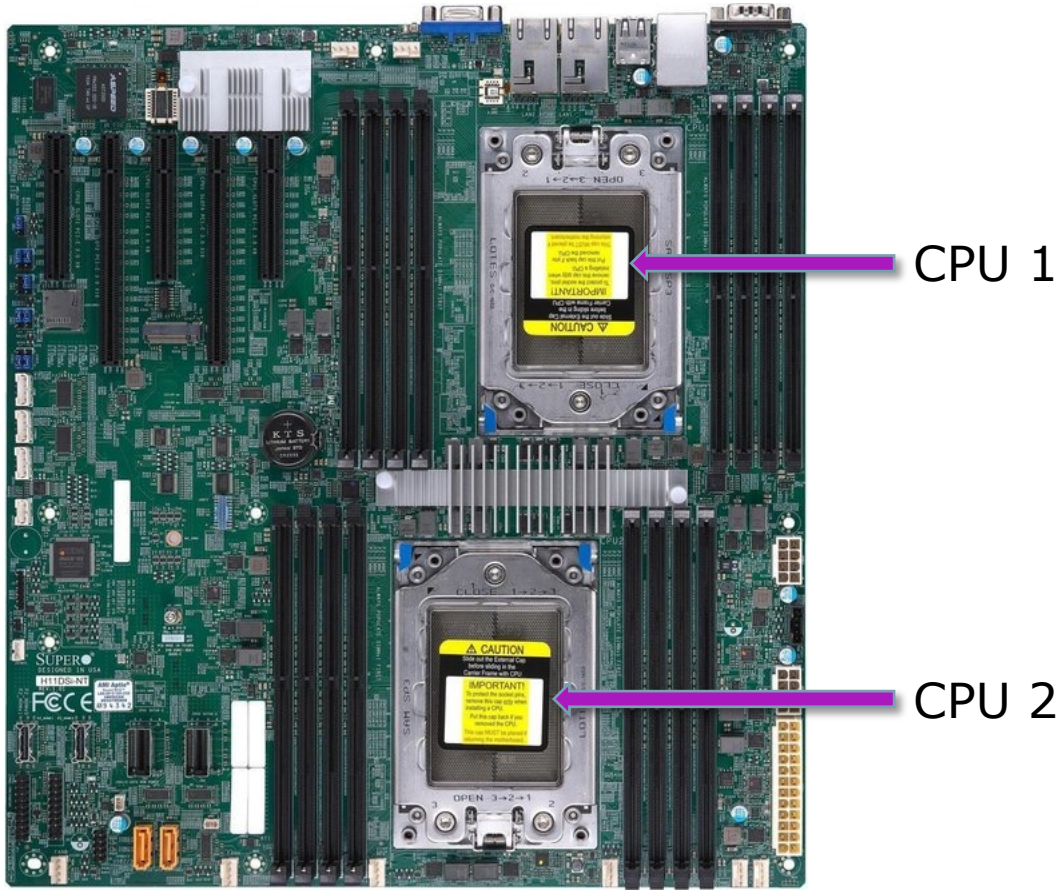
There is no guarantee they'll ever both be running at the same time

Parallelism

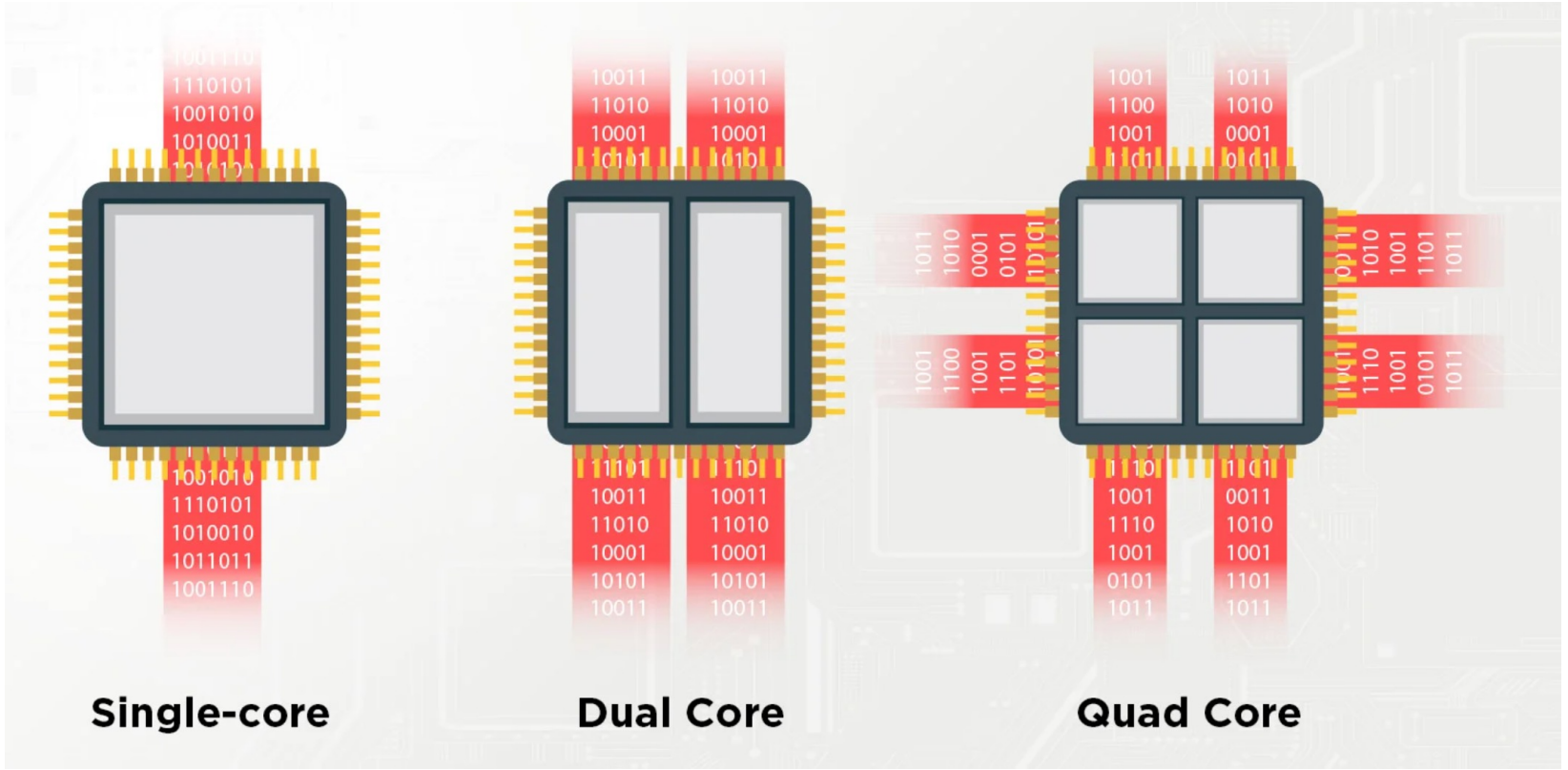
Two or more tasks that run *at the same time*

High-Level Concurrency

Machine-Level Concurreny



Processor-Level Concurrency



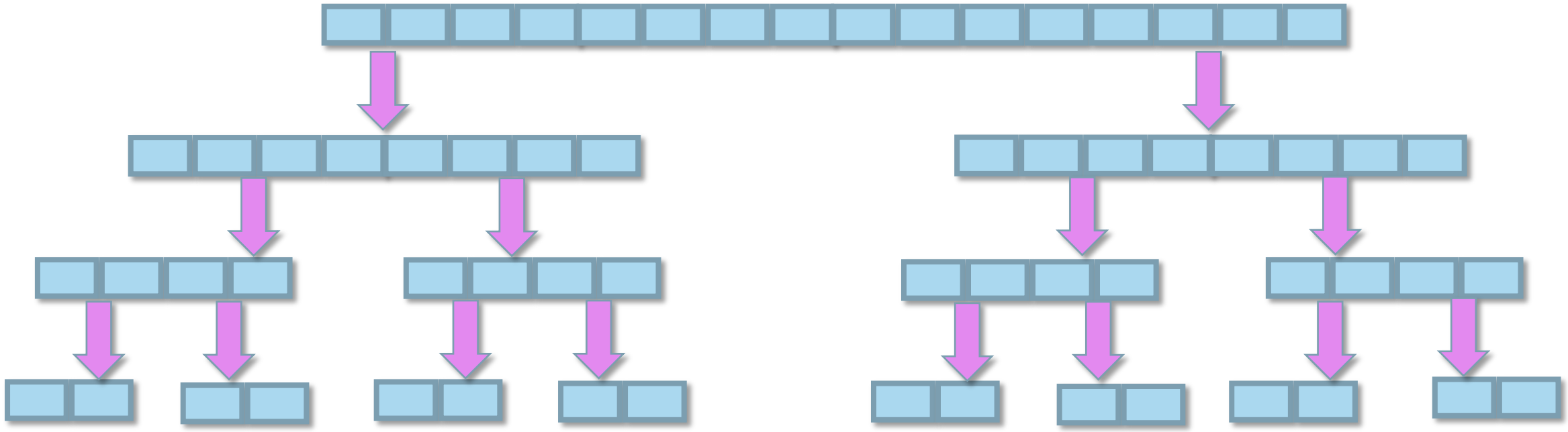
Concurrent Programming In Java

- Java provides several different approaches to concurrent programming
- The hardest way is to use the Thread class
  - Manually code splitting data, processing and recombining data

- Java has introduced many libraries to make this easier
  - JSR-166 Concurreny utilities
  - Semaphore, mutex, etc.

Concurrent Programming in Java

- JDK 7 introduced the fork-join framework
- Recursively decompose a task into smaller subtasks



Concurrent Programming In Java

- JDK 8 introduced streams and Lambda expressions to Java
  - Functional style of programming
- Library call hides complexities of concurrency
  - Uses fork-join framework
- But can deliver non-deterministic results

```
List<Integer> values = getData();

values.stream()
    .mapToInt(Integer::intValue)
    .sum();

values.parallelStream()
    .mapToInt(Integer::intValue)
    .sum();
```

Concurrent Programming In Java

- Don't assume that doing things concurrently will do things faster...
- ...or more efficiently

- Some operations decompose well to concurrent operations: `sum()`, `min()`, `max()`
- Some operations do not: `sort()`

- Do not use parallel streams for things that cannot easily be split or do IO
  - e.g. processing lines being read from a file (inherently sequential)

- Concurrent programming can use multiple execution units (CPUs, cores) to *possibly* improve performance
  - But does not require them

azul

Some Background on Vectors and SIMD

Vector Mathematics Using Matrices

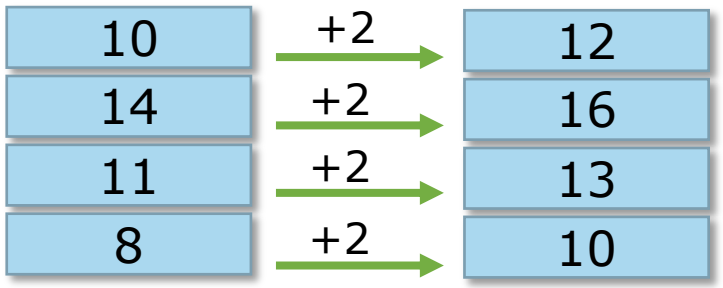
- Examples

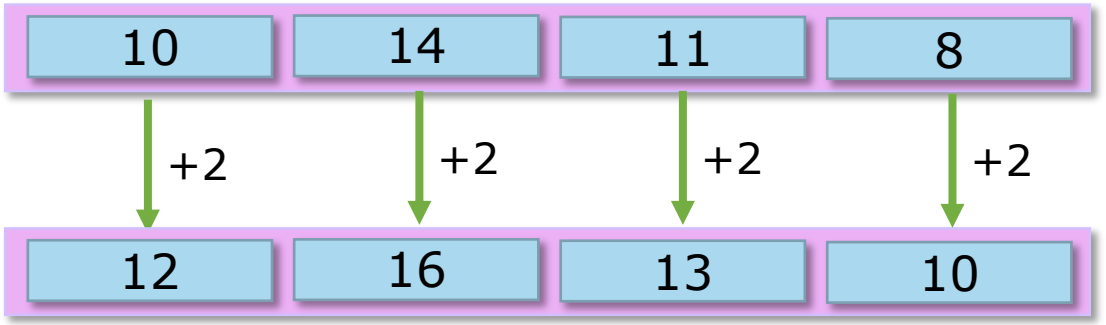
$$[\ 4, 3, 7\ ] + [\ 3, 5, 2\ ] = [\ 7, 8, 9\ ]$$

$$[\ 2, 3, 4\ ]\ X \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix} = 29$$

Single Instruction Multiple Data (SIMD)

- SIMD enables parallel processing *within* a single execution unit
- Uses very wide registers to hold multiple values
  - Each register has a number of lanes





Single Instruction Multiple Data (SIMD)

- First introduced in 1966

ILLIAC IV

Single Instruction Multiple Data (SIMD)

- Supercomputers of the 1970s

CDC Star-100

Texas Instruments ASC

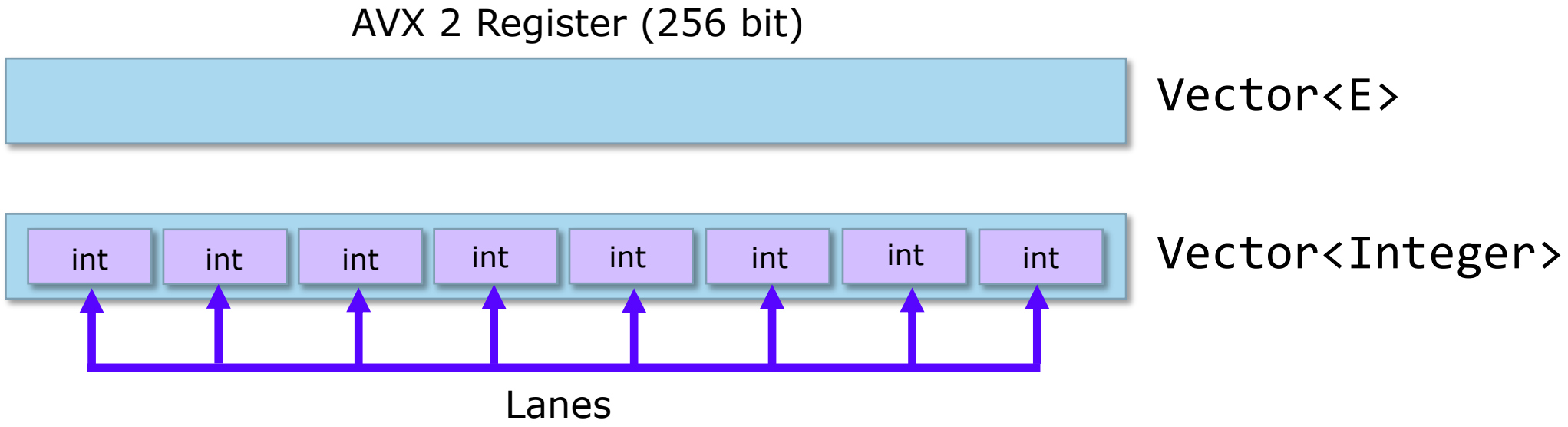
SIMD Instruction History

- 1994: HP PA-RISC processor, MAX instructions (32-bit and 64-bit registers)
- 1995: Sun Microsystems UltraSPARC processor, VIS instructions (64-bit registers)
- 1996: Intel Pentium P5 processor, MMX instructions (64-bit registers)
- 1999: Intel Pentium III processor, SSE intsructions (128-bit registers)
- 2011: Intel SandyBridge processor, AVX instructions (128-bit registers)
- 2013: Intel Haswell processor, AVX-2 instructions (256-bit registers)
- 2016: Intel Xeon Phi processor, AVX 512 instructions (512-bit registers)

azul

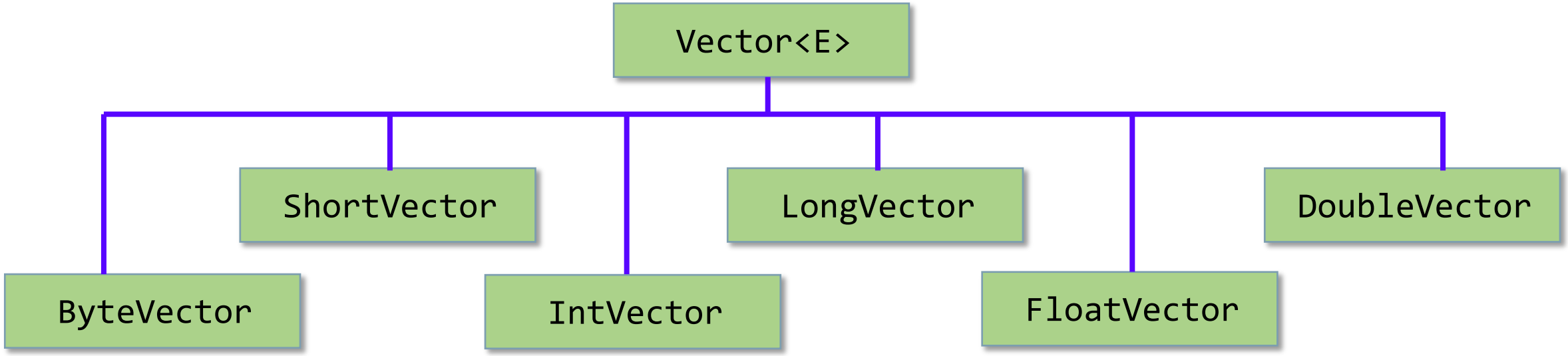
The Vector API (JDK 21, Sixth Incubator)

Vectors In Java



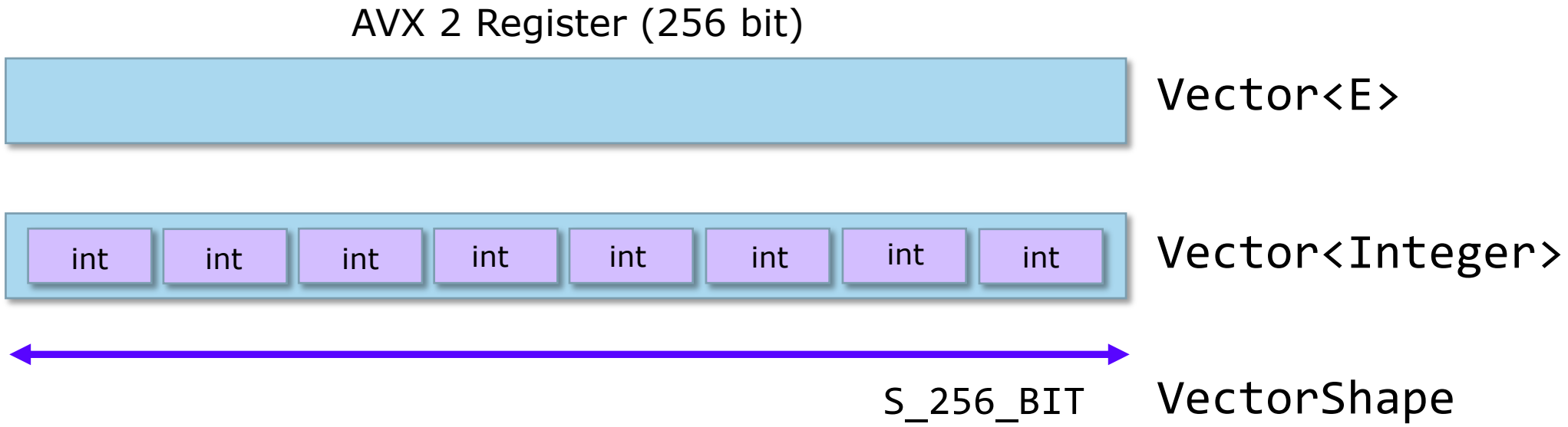
- `Vector` includes methods for operations that are common to all vector types
  - `add()`, `sub()`, `div()`, `mul()`, `min()`, `max()`, etc.

Vectors In Java



- Adds methods for operations that are type-specific
  - e.g. `fromArray()`

Vectors In Java




```
VectorShape.S_64_BIT
VectorShape.S_128_BIT
VectorShape.S_256_BIT
VectorShape.S_512_BIT
```

```
VectorShape.S_Max_BIT
```

Vector Shape

- Use `S_Max_BIT` to represent the biggest register on the machine currently being used
  - Better cross-platform code
- Intel processors can support more than one shape
  - ARM processors only support one shape (`S_Max_BIT`), which can be 128 to 2048 bits (vendor specific)

Vector Species

- A `Vector` has a species, represented by `VectorSpecies<E>`
- This is a combination of the **element type** and the **shape**
- The VectorSpecies encapsulates three pieces of information:

Species[ETYPE, VLENGTH, SHAPE] e.g. Species[int, 8, S_256_BIT]

  - ETYPE: Primitive lane type
  - VLENGTH: Vector lane count
  - SHAPE: The vector shape

Vector Species

- Simplifies determining aspects of how to use Vectors on a specific machine

```
private final static VectorSpecies<Integer> INT_SPECIES = IntVector.SPECIES_256;
private final int[] arrayA = new int[1_000_000];
private final int[] arrayB = new int[1_000_000];

...
int vectorLoopSize = INT_SPECIES.loopBound(arrayA.length);
int numberOfLanes = INT_SPECIES.length();

for (int i = 0; i < vectorLoopSize; i += numberOfLanes) {
  IntVector va = IntVector.fromArray(INT_SPECIES, arrayA, i);
  IntVector vb = IntVector.fromArray(INT_SPECIES, arrayB, i);

  // Use the vectors
}
```

Vector Mask

- When processing an array, the length may not divide cleanly by the number of lanes
- Example:
  - An int array has 30 elements and we're using our 256 bit example vector
  - The loop stride length is 8 (number of vector lanes)
  - The last iteration only requires 6 of the 8 lanes
- We can use a mask to suppress the operation on the last two lanes of the vector in the final iteration
- Depending on what type of operation is being performed the mask will have different effects

> *"An operation suppressed by a mask will never cause an exception or side effect of any sort, even if the underlying scalar operator can potentially do so."*

Vector Mask

```
for (int i = 0; i < vectorLoopSize; i += numberOfLanes) {
  IntVector va = IntVector.fromArray(INT_SPECIES, arrayA, i);
  IntVector vb = intVector.fromArray(INT_SPECIES, arrayB, i);

  // Use the vectors
}
```

Vector Mask

```
for (int i = 0; i < vectorLoopSize; i += numberOfLanes) {
  VectorMask<Integer> mask = INT_SPECIES.indexInRange(i, arrayA.length);
  IntVector va = IntVector.fromArray(INT_SPECIES, arrayA, i, mask);
  IntVector vb = IntVector.fromArray(INT_SPECIES, arrayB, i, mask);

  // Use the vectors
}
```

Vector Mask

- A mask can also be used to perform conditional execution for a vector
  - If a lane is set, the value comes from the second vector passed as a parameter
  - If a lane is unset, the value comes from the original vector

```
VectorMask<Integer> mask = createMask();

IntVector va = getVectorA();
IntVector vb = getVectorB();

IntVector combined = va.blend(vb, mask);
```

VectorShuffle

- Is used to change the order of the elements in a vector

```
IntVector va = getVectorA();
int[] shuffleValues = {1, 3, 5, 7, 0, 2, 4, 6};

VectorShuffle<Integer> shuffle =
    VectorShuffle.fromValues(INT_SPECIES, shuffleValues);

va.rearrange(shuffle);
```

Vector Operators

- What can we do with our Vectors?

- Types of operators
  - **Associative**: order does not matter (e.g. ADD, AND)
  - **Binary**: Takes two arguments (e.g. DIV, POW)
  - **Comparison**: (e.g. EQ, NE, LT)
  - **Conversion**: (e.g. D2B, B2F, F2D)
  - **Ternary**: Takes three arguments (e.g. FMA)
  - **Test**: (e.g. IS_FINITE, IS_NEGATIVE)
  - **Unary**: Takes one argument (e.g. LOG, NEG, NOT)

- See the `VectorOperators` class for the full llist

Mathematical Operations

```
public float[] hypo(float[] a, float[] b) {
  float[] result = new float[a.length];

  for (int i = 0; i < a.length; i += FLOAT_SPECIES.length()) {
    var mask = FLOAT_SPECIES.indexInRange(i, a.length);
    var va = FloatVector.fromArray(FLOAT_SPECIES, a, i, mask);
    var vb = FloatVector.fromArray(FLOAT_SPECIES, b, i, mask);
    var vc = va.mul(va).add(vb.mul(vb)).sqrt();
    vc.intoArray(result, i);
  }

  return result;
}
```

Reduction Operations

- Reduce the lanes of a vector to a single value
- Eaxample: add all the values to get a total

```
public double average(int[] arr) {
  double sum = 0;

  for (int i = 0; i< arr.length; i += INT_SPECIES.length()) {
    var mask = INT_SPECIES.indexInRange(i, arr.length);
    var v = IntVector.fromArray(INT SPECIES, arr, i, mask);
    sum += v.reduceLanes(VectorOperators.ADD, mask);
  }

  return sum / arr.length;
}
```

azul

How Well Does It Work?

SIMD Is Fast, Yes

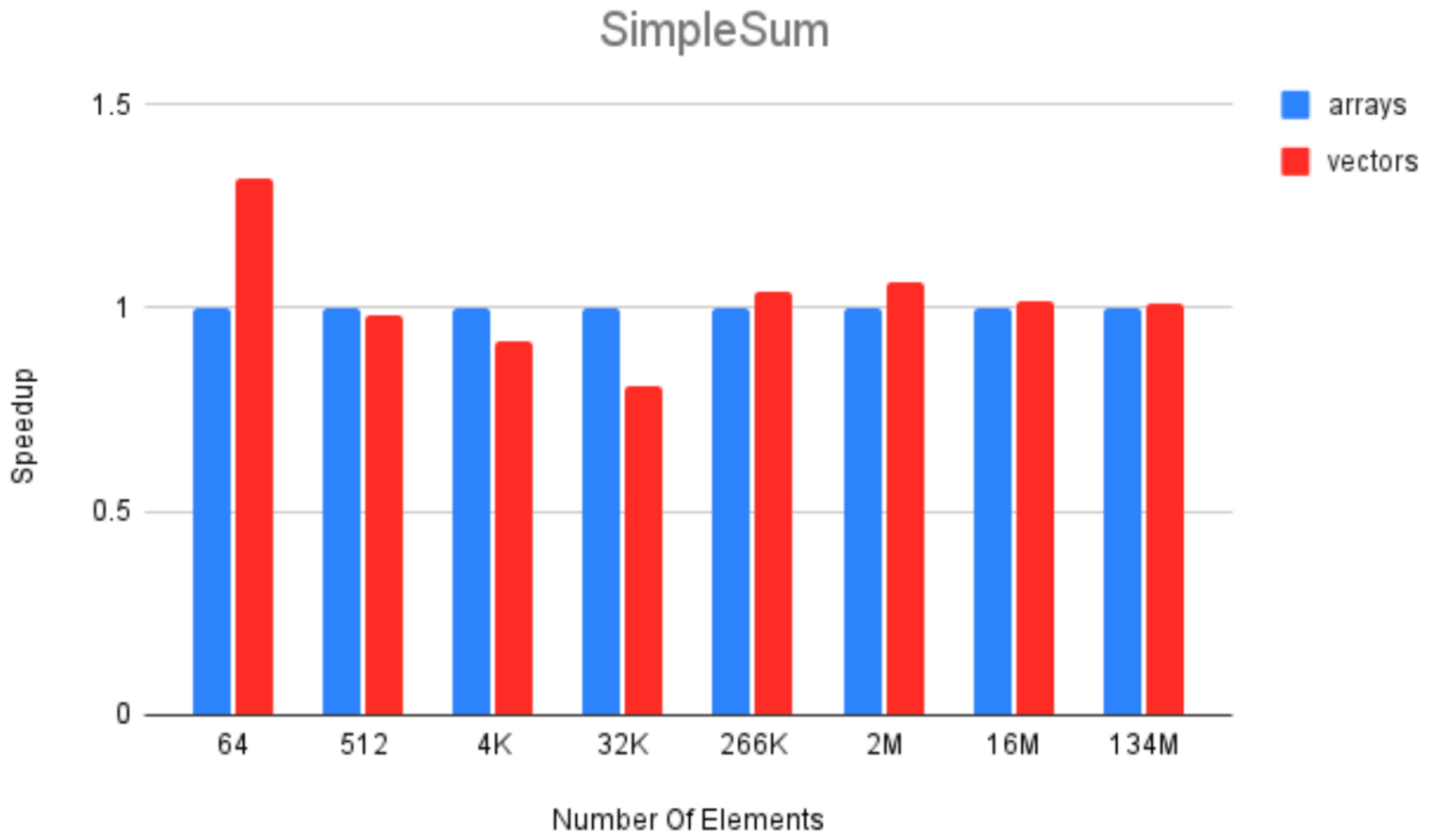
- But...
- Remember that main memory access is (relatively) slow
- Larger arrays will not fit in the CPU cache
  - L1 cache: 3-5 cycles
  - L2 cache: 8-20 cycles
  - L3 cache: 50-80 cycles
  - RAM: 100+ cycles
- The further you go for your data the less improvements using SIMD will provide

Auto Vectorization

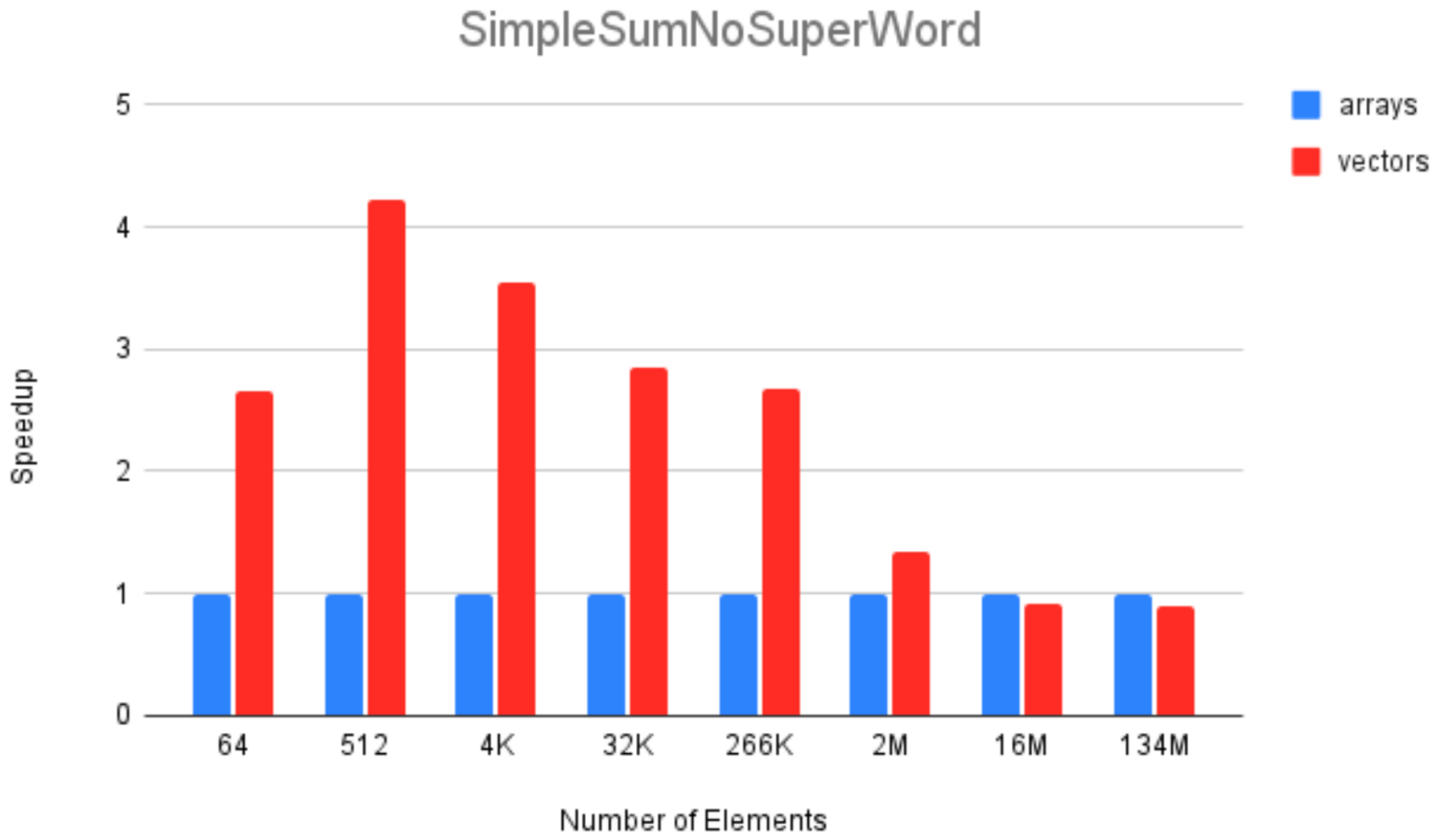
- The JIT compiler already includes the ability to use SIMD instructions where available
- For many situations this will deliver equivalent (or even better) results
- This becomes apparent if you turn off auto-vectorization
  - `-XX:-UseSuperWord`

- The following results are from Tomer Zeltzer
  - medium.com/@tomerr90/

Simple Sum (Adding Two Vectors)



Simple Sum (Adding Two Vectors)



Array Statistics

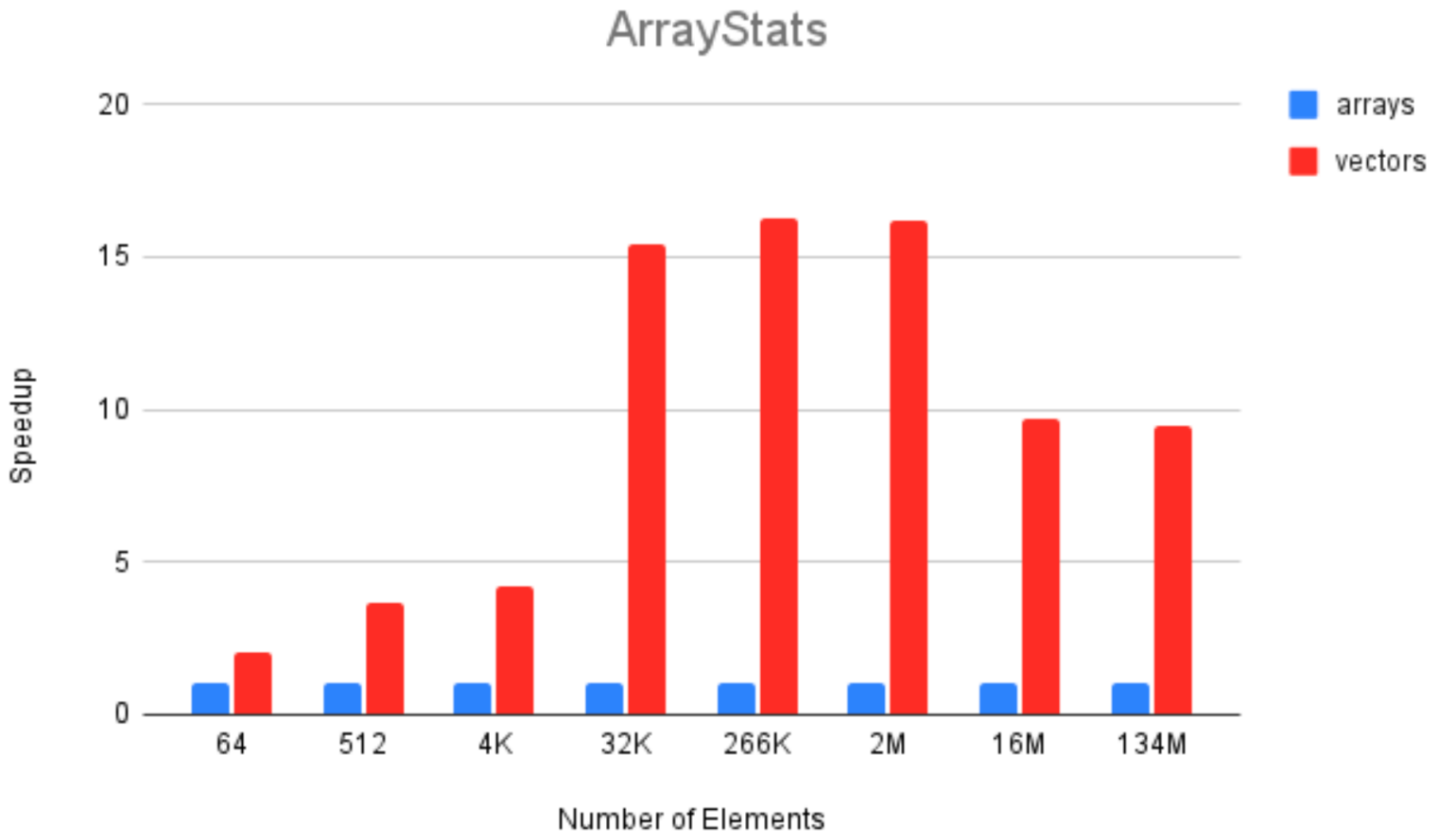
- Let's introduce a conditional
- How many elements of an array are less than, equal to and greater than the elements of a second array

```
int lt = 0, eq = 0; gt = 0;

for (int i = 0; i < arraySize; i += SPECIES.length()) {
    FloatVector aVector = FloatVector.fromArray(SPECIES, a, i);
    FloatVector bVector = FloatVector.fromArray(SPECIES, b, i);

    lt += aVector.lt(bVector).trueCount();
    eq += aVector.eq(bVector).trueCount();
    gt += aVector.gt(bVector).trueCount();
}
```

Array Statistics



Branches (Can) Slow You Down

```
int[] arrayA;
int[] arrayB;

for (int i = 0; i < arrayA.length; i++) {
    if (arrayB[i] & 0x1 == 0)
        arrayA[i] += arrayB[i]
}
```

HotSpot C2 JIT

Per element jumps
2 elements per
iteration

		0x3001067f	addl %ecx, 12(%rsi)	0x014e0c
		0x30010682	movl $1, %edi	0xbf01000000
		0x30010687	cmpl $1, %eax	0x83f801
		0x3001068a	je 56 ; ABS: 0x300106c4	0x7438
		0x3001068c	subq %rdi, %rax	0x4829f8
		0x3001068f	leaq 16(%rdx,%rdi,4), %rcx	0x488d4cba10
		0x30010694	leaq 16(%rsi,%rdi,4), %rdx	0x488d54be10
		0x30010699	nopl (%rax)	0x0f1f8000000000
16.84%	1,286	0x300106a0	movl -4(%rcx), %esi	0x8b71fc
6.10%	466	0x300106a3	testb $1, %sil	0x40f6c601
		0x300106a7	jne 3 ; ABS: 0x300106ac	0x7503
7.61%	581	0x300106a9	addl %esi, -4(%rdx)	0x0172fc
29.41%	2,246	0x300106ac	movl (%rcx), %esi	0x8b31
2.25%	172	0x300106ae	testb $1, %sil	0x40f6c601
		0x300106b2	jne 2 ; ABS: 0x300106b6	0x7502
8.00%	611	0x300106b4	addl %esi, (%rdx)	0x0132
29.73%	2,271	0x300106b6	addq $8, %rcx	0x4883c108
		0x300106ba	addq $8, %rdx	0x4883c208
		0x300106be	addq $-2, %rax	0x4883c0fe
		0x300106c2	jne -36 ; ABS: 0x300106a0	0x75dc
0.03%	2	0x300106c4	addq $24, %rsp	0x4883c418
0.03%	2	0x300106c8	retq	0xc3
		0x300106c9	movq %rsi, 16(%rsp)	0x4889742410
		0x300106ce	movq %rdx, 8(%rsp)	0x4889542408
		0x300106d3	movabsq $805334400, %rax	0x48b8806d003000000000
		0x300106dd	callq *%rax	0xffd0

```
int[] arrayA;
int[] arrayB;

for (int i = 0; i < arrayA.length; i++) {
    if (arrayB[i] & 0x1 == 0)
        arrayA[i] += arrayB[i]
}
```

Falcon JIT

		0x3001455b	movq %rdi, %rbx
		0x3001455e	nop
0.15%	4	0x30014560	vmovdqu -96(%r11), %ymm2
12.31%	320	0x30014566	vmovdqu -64(%r11), %ymm3
0.50%	13	0x3001456c	vmovdqu -32(%r11), %ymm4
2.04%	53	0x30014572	vmovdqu (%r11), %ymm5
0.31%	8	0x30014577	vpand %ymm0, %ymm2, %ymm6
4.54%	118	0x3001457b	vpand %ymm0, %ymm3, %ymm7
0.69%	18	0x3001457f	vpand %ymm0, %ymm4, %ymm8
1.35%	35	0x30014583	vpand %ymm0, %ymm5, %ymm9
0.42%	11	0x30014587	vpcmpeqd %ymm1, %ymm6, %ymm6
2.58%	67	0x3001458b	vpmaskmovd -96(%rcx), %ymm6, %ymm10
3.58%	93	0x30014591	vpcmpeqd %ymm1, %ymm7, %ymm7
2.12%	55	0x30014595	vpmaskmovd -64(%rcx), %ymm7, %ymm11
12.12%	315	0x3001459b	vpcmpeqd %ymm1, %ymm8, %ymm8
1.50%	39	0x3001459f	vpmaskmovd -32(%rcx), %ymm8, %ymm12
3.69%	96	0x300145a5	vpcmpeqd %ymm1, %ymm9, %ymm9
1.81%	47	0x300145a9	vpmaskmovd (%rcx), %ymm9, %ymm13
12.27%	319	0x300145ae	vpaddd %ymm2, %ymm10, %ymm2
0.58%	15	0x300145b2	vpaddd %ymm3, %ymm11, %ymm3
0.19%	5	0x300145b6	vpaddd %ymm4, %ymm12, %ymm4
0.58%	15	0x300145ba	vpaddd %ymm5, %ymm13, %ymm5
3.27%	85	0x300145be	vpmaskmovd %ymm2, %ymm6, -96(%rcx)
7.15%	186	0x300145c4	vpmaskmovd %ymm3, %ymm7, -64(%rcx)
13.65%	355	0x300145ca	vpmaskmovd %ymm4, %ymm8, -32(%rcx)
4.58%	119	0x300145d0	vpmaskmovd %ymm5, %ymm9, (%rcx)
6.81%	177	0x300145d5	subq $-128, %r11
0.69%	18	0x300145d9	subq $-128, %rcx
0.31%	8	0x300145dd	addq $-32, %rbx
		0x300145e1	jne -135 ; ABS: 0x30014560
		0x300145e7	testl %r9d, %r9d
		0x300145ea	jne -356 ; ABS: 0x3001448c

Using AVX2 vector instructions
32 elements per iteration

Broadwell E5-2690-v4

azul

Summary

Conclusions

- Vector API allows explicit programming of SIMD processing
- Simple situations will not necessarily benefit
  - JIT compiler will use SIMD automatically
- More complex situations that are not recognisable by the JIT will benefit
- Benefits are reduced as the size of arrays increases
  - Cache and memory latency becomes increasingly significant

- Ideally, the compiler would autovectorise all our code

Azul Java

- Platform Core Builds of OpenJDK provides
  - TCK tested binaries (not JDK 6 or 7)
  - Updates within a time defined SLA
  - All security patches and bug fixes for all updates
  - CPUs and PSUs
  - 24x7 support issue reporting

- Platform Prime Builds of OpenJDK provide
  - C4 fully concurrent, pauseless garbage collection
  - Falcon C2 JIT compiler replacement, delivering better optimisations (including enhanced use of vectors)
  - ReadyNow JIT compiler profile to reduce warmup time

Thank You.

Simon Ritter, Deputy CTO
sritter@azul.com

@speakjava