# SKILLPILLS

## Skill Pill: Julia
### Lecture 4: Distributed and parallel computing

Valentin Churavy   James Schloss

Okinawa Institute of Science and Technology
*valentin.churavy@oist.jp*
*james.schloss@oist.jp*

July 13, 2017

OIST

### Necessary packages

- ► SIMD.jl
- ► MPI.jl
- ► DistributedArrays.jl
- ► CUDAnative.jl if your computer has a NVidia GPU

- Instruction level parallelism
- Shared-memory and threading
- Distributed
- Accelerators e.g.GPGPU

```
function padd(a, b, x, y)
  c = a + b
  z = x + y
  return c, z
end
```

### Observation

The two operations are independent of each other and we could execute them in parallel.

1. Use `@code_llvm` and `@code_native` to understand what is happening
2. Establish a baseline performance with `@benchmark`
3. Start Julia with `julia -O3`
4. Compare the llvm and native code and your benchmark results
5. Note that there is next to no performance benefit in this example, but that changes once you scale up

```
function sum(x)
  acc = 0.0
  for i in 1:length(a)
    acc += x
  end
end
```

## Observation

Is each loop iteration independent from each other? Yes and no. Standard addition is associative and the order of operations has no impact. Floating-point addition is non-associative and the order of operations is important. The compiler cannot vectorise this loop, without changing the semantics.

1. Learn about @inline and @simd

# Explicit SIMD



## SIMD.jl

Instead of relying on the compiler to optimise and vectorise our code correctly we can also write explicit SIMD code.

```julia
using SIMD
function add(out::Vector{Float64}, x::Vector{Float64},
    y::Vector{Float64})
  # My laptop supports AVX 256bit 4xFloat64
  @assert length(x) % 4 == 0
  for i in 1:4:length(x)
    vx = vload(Vec{4, Float64}, x, i)
    vy = vload(Vec{4, Float64}, y, i)
    vo = vx + vy
    vstore(vo, out, i)
  end
end
```

Explain current fork join model and caveats

Simple example

complex example with loop splitting, random etc...

Atomics

MPI

DistibutedArrays