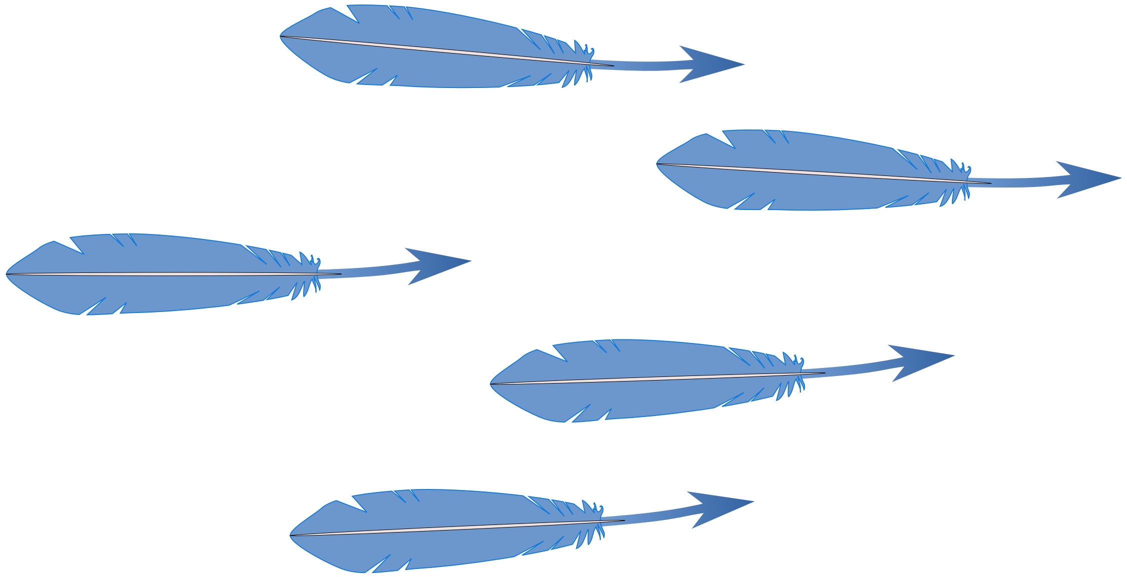
Speeding up VecTcl — experiments with compilation to machine code



Christian Gollwitzer

EuroTcl 2015

What is VecTcl?

Tcl has (scalar) math in the core:

$$x = \frac{1}{2a}\left(b \pm \sqrt{b^2 - 4ac}\right)$$

```
set x [expr {($b+sqrt($b**2-4*$a*$c))/(2*$a)}]
```

There is no direct support for vector math:

$$x = \vec{a} \cdot \vec{b} = \sum_i a_i b_i$$

```
set x 0.0
foreach ai $a bi $b {
  set x [expr {$x+$a*$b}]
}
```

VecTcl:

```
vexpr { x=a*b‘ }
```

Linear regression

Math

$$
\begin{aligned}
\bar{x} &= \frac{1}{N}\sum x_i \\
\bar{y} &= \frac{1}{N}\sum y_i \\
\beta &= \frac{\sum_i (x_i - \bar{x}) \cdot (y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \\
\alpha &= \bar{y} - \beta\bar{x}
\end{aligned}
$$

Tcl

```
set xsum 0.0; set ysum 0.0
foreach x $xv y $yv {
    set xsum [expr {$xsum + $x}]
    set ysum [expr {$ysum + $y}]
}
set xm [expr {$xsum/[llength $xv]}]
set ym [expr {$ysum/[llength $xv]}]
set xsum 0.0; set ysum 0.0
foreach x $xv y $yv {
    set dx [expr {$x - $xm}]
    set dy [expr {$y - $ym}]
    set xsum [expr {$xsum + $dx * $dy}]
    set ysum [expr {$ysum + $dx * $dx}]
}
set b [expr {$xsum / $ysum}]
set a [expr {$ym - $b * $xm}]
```

VecTcl

```
vexpr {
  xm=mean(xv)
  ym=mean(yv)
  beta=sum((xv-xm).*(yv-ym))./sum((xv-xm).^2)
  alpha=ym-beta*xm
}
```

☺ Much easier

☺ Faster (mostly)

How does it work?

VecTcl is a 2-layered system

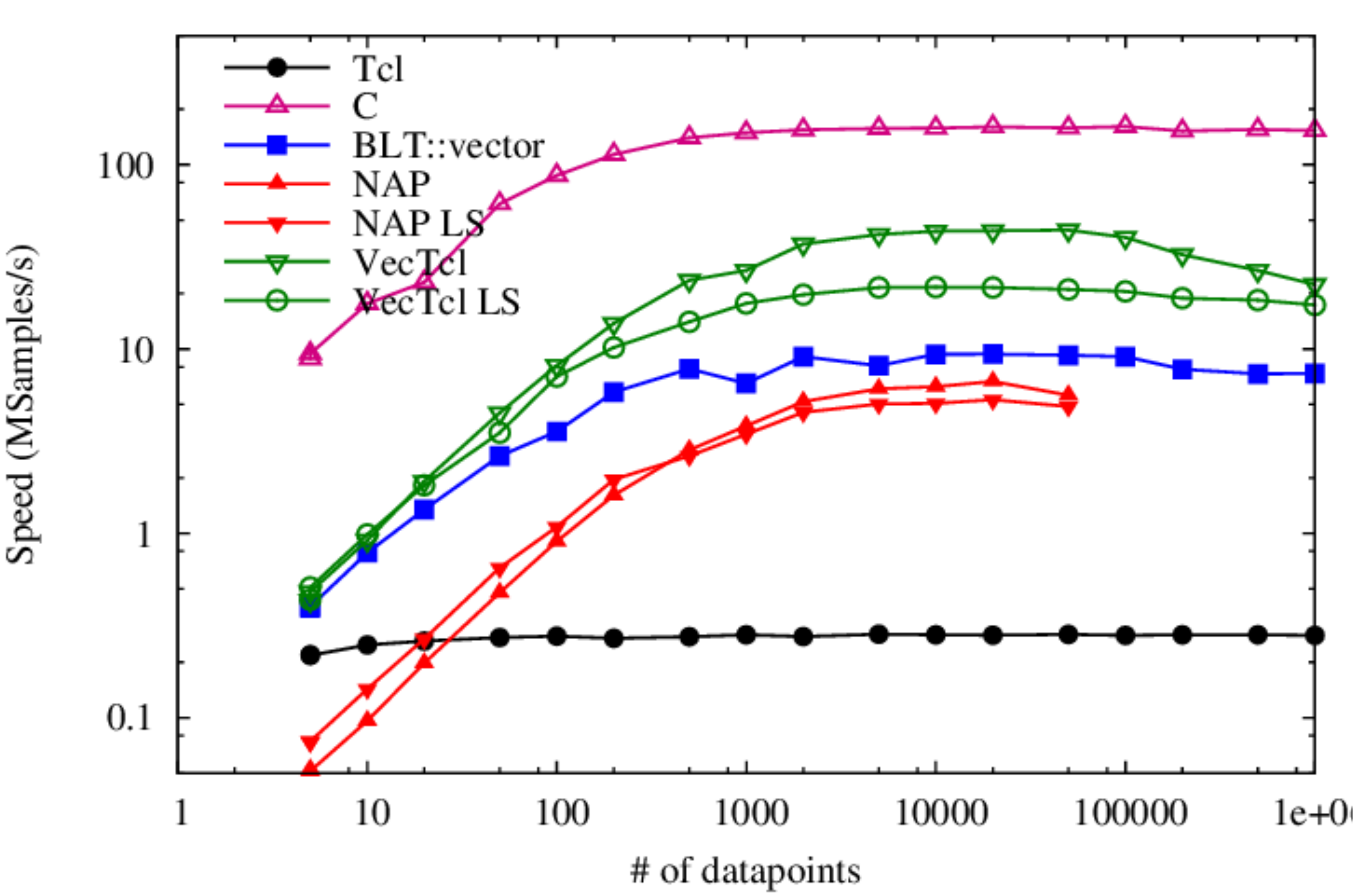
```
vexpr {
    a={1 2 3}
    c=2*(a+{4 5 6})
}
```

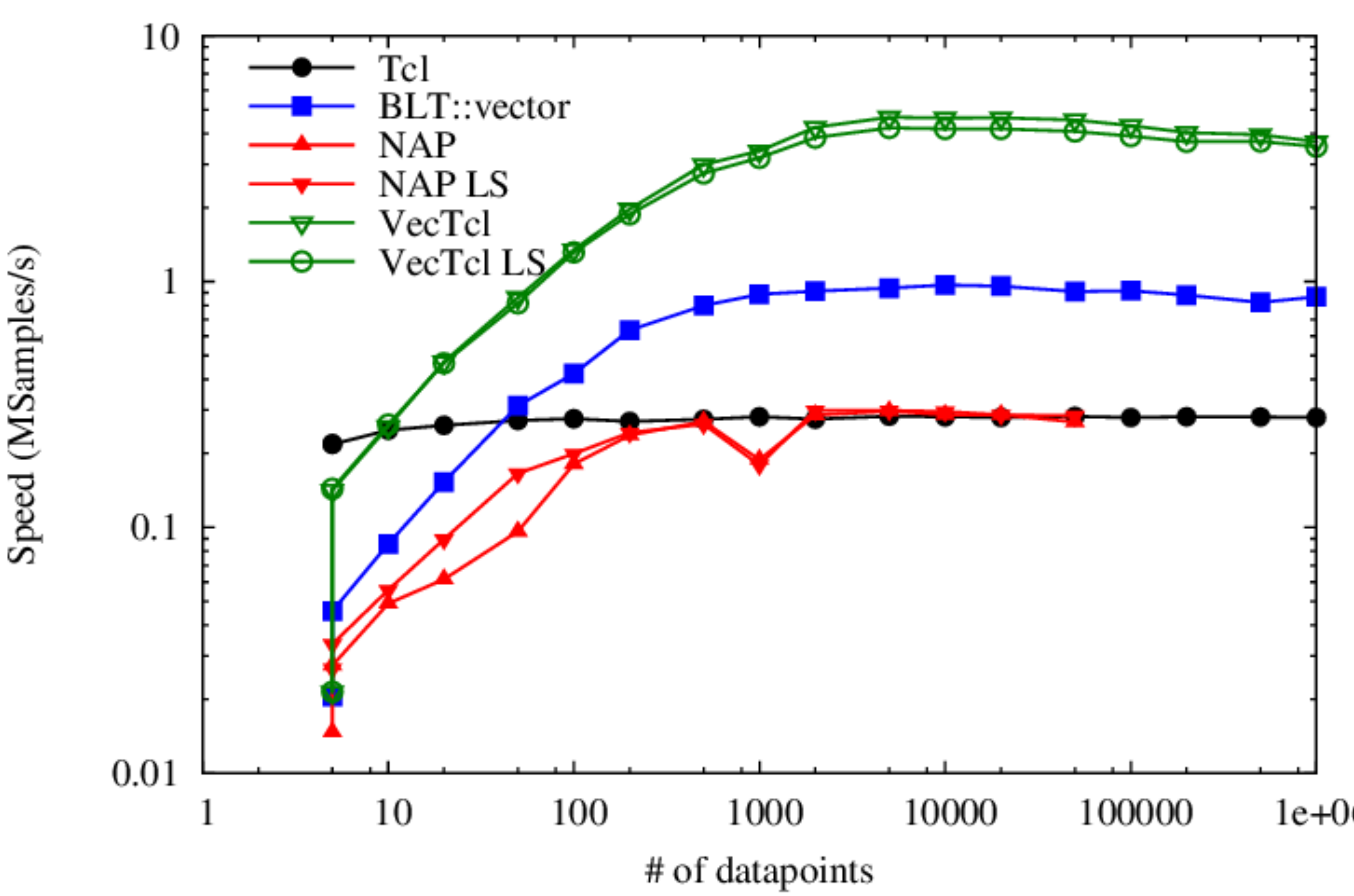
Compiler, written in Tcl

Runtime, written in C

```
proc numarray::compiledexpressionXX {} {
  upvar 1 a a
  upvar 1 c c
  set a {1 2 3}
  set c [numarray::* 2 [numarray::+ [set a] {4 5 6}]]
}
numarray::compiledexpressionXX
```

Benchmarks - linear regression


Only computation

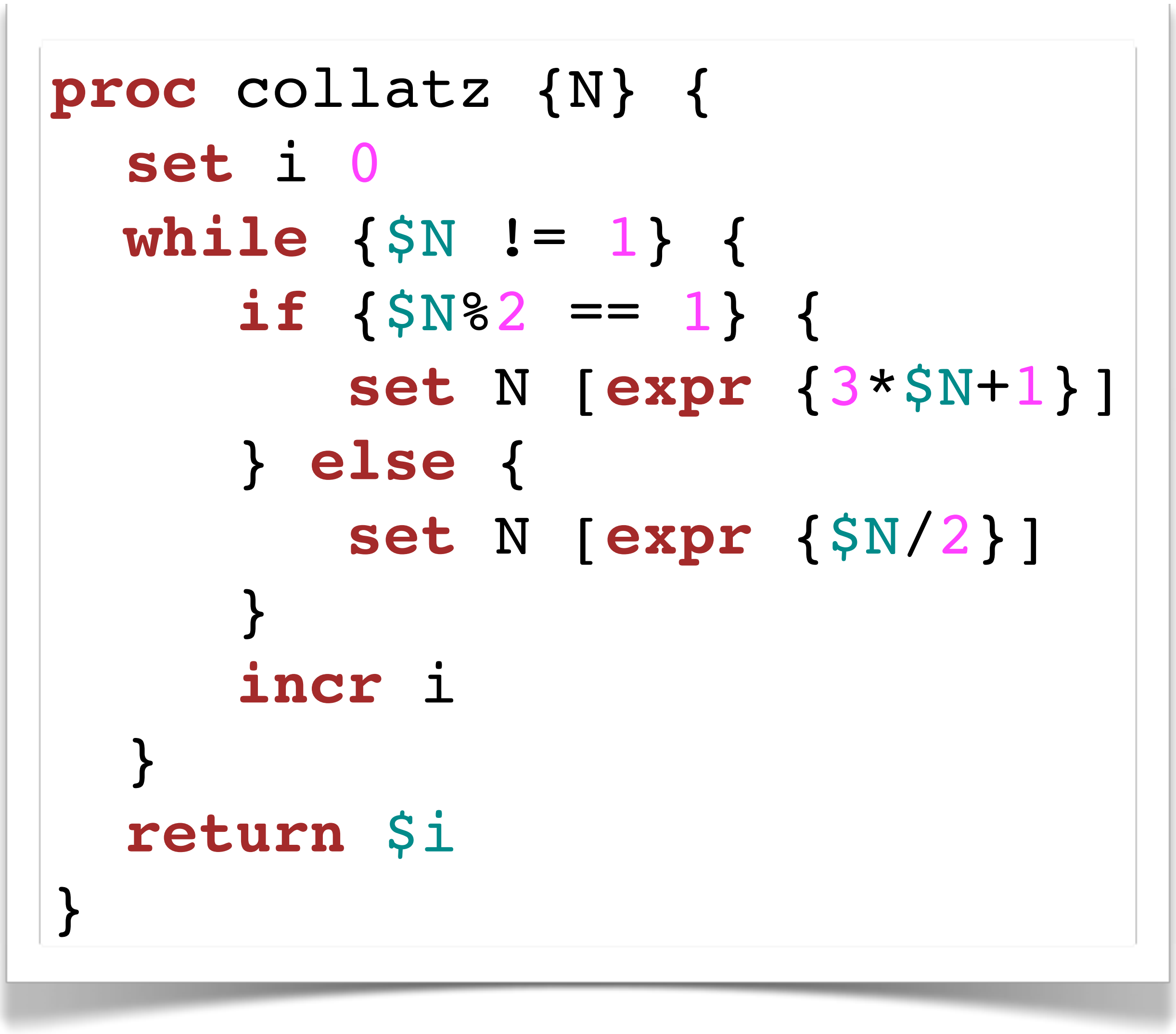

Total (setup + computation)

- VecTcl is 4x slower than C, but still faster than the others
- Shimmering is 5x slower than actual computation
- Competitors are still slower there

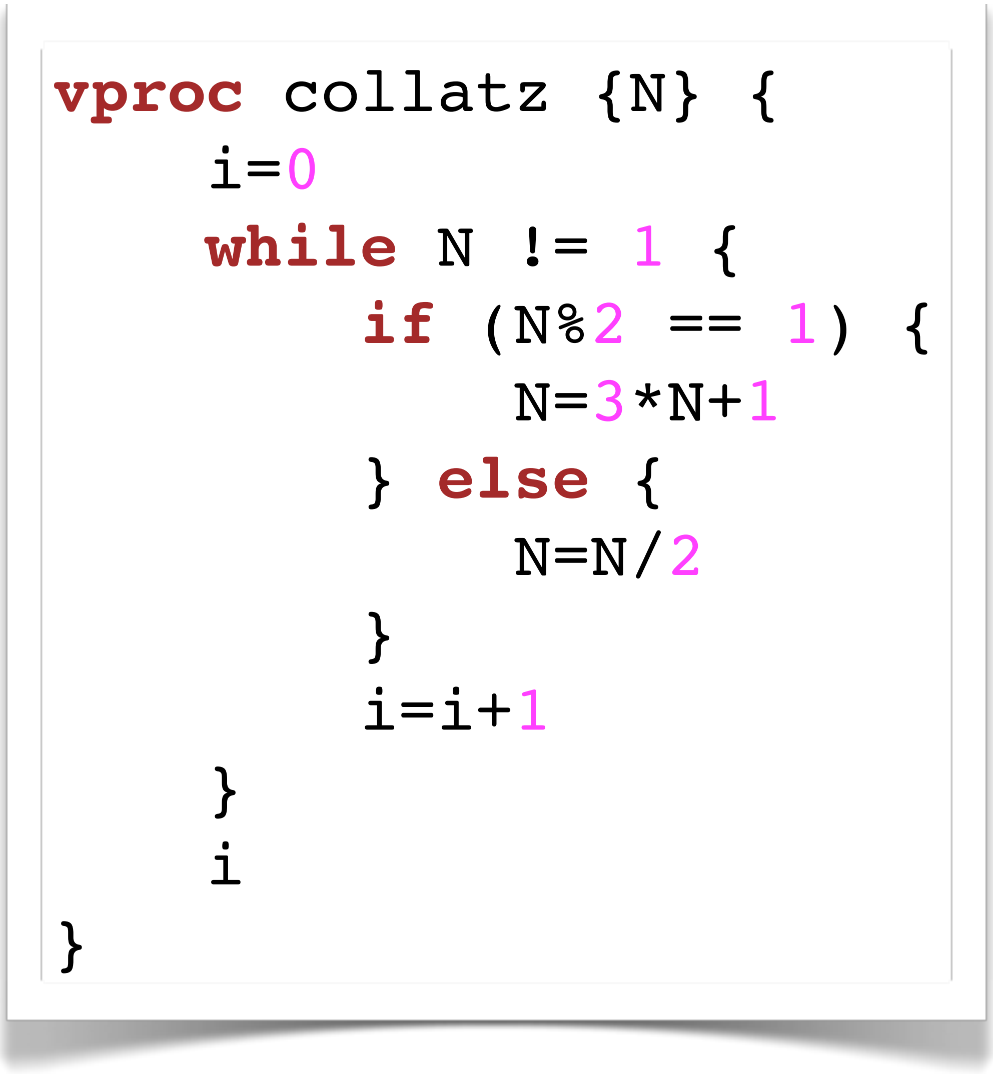
Live Demo

VecTcl sucks at...

Scalar math



```
proc collatz {N} {
  set i 0
  while {$N != 1} {
     if {$N%2 == 1} {
        set N [expr {3*$N+1}]
     } else {
        set N [expr {$N/2}]
     }
     incr i
  }
  return $i
}
```



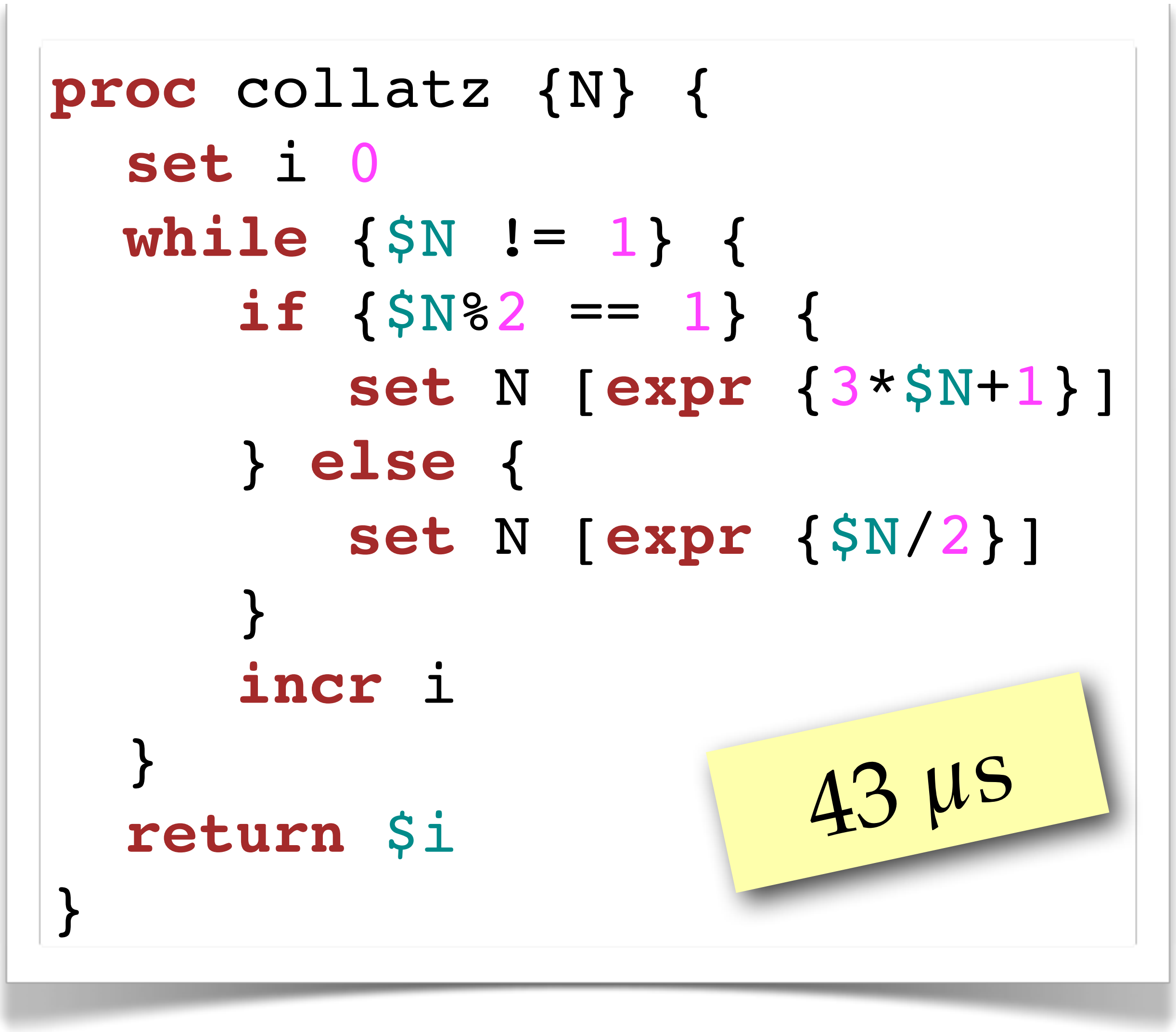
```
vproc collatz {N} {
    i=0
    while N != 1 {
        if (N%2 == 1) {
            N=3*N+1
        } else {
            N=N/2
        }
        i=i+1
    }
    i
}
```

- Bytecoded by Tcl
- No function call
- Dynamic data types

- Vectors used as scalars
- 4.5 function calls per iteration

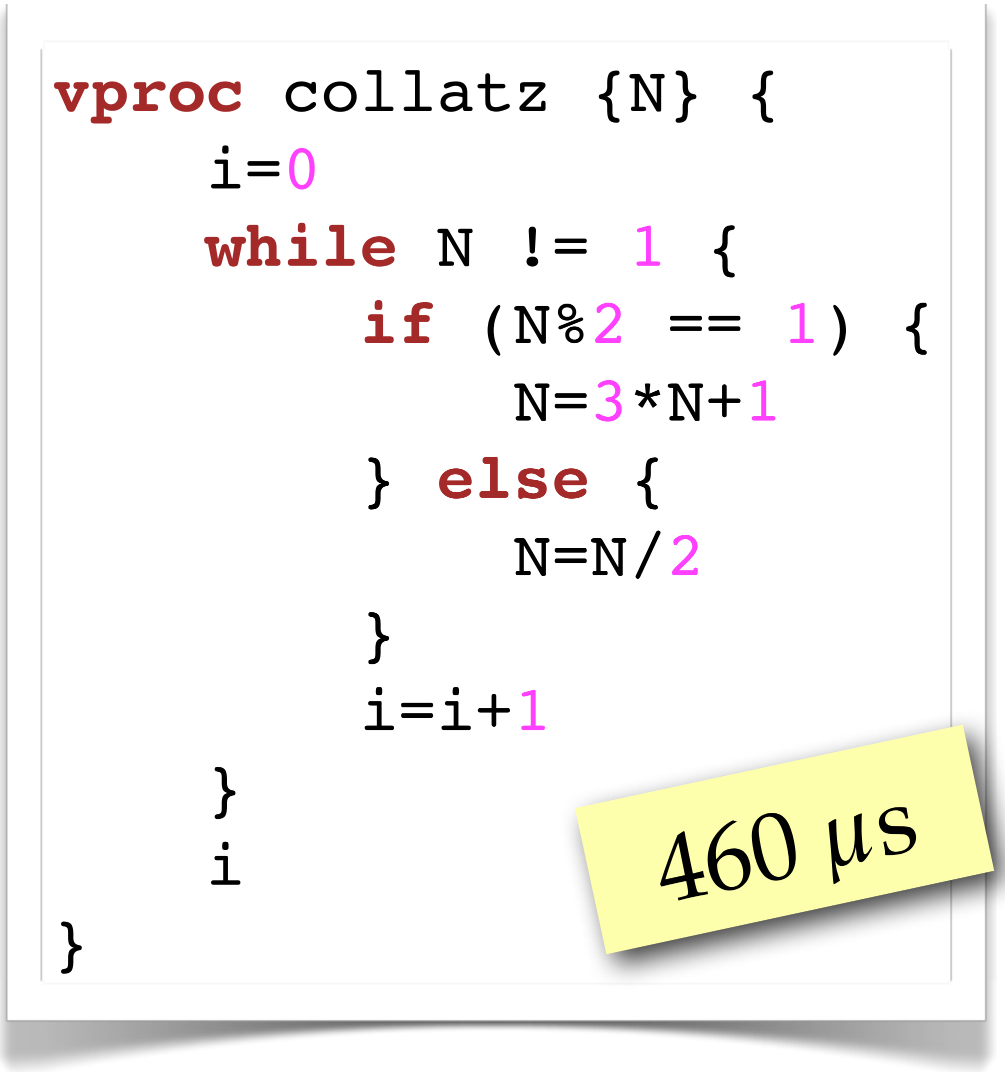
VecTcl sucks at...

Scalar math



```
proc collatz {N} {
  set i 0
  while {$N != 1} {
     if {$N%2 == 1} {
        set N [expr {3*$N+1}]
     } else {
        set N [expr {$N/2}]
     }
     incr i
  }
  return $i
}
```

43 µs



```
vproc collatz {N} {
    i=0
    while N != 1 {
        if (N%2 == 1) {
            N=3*N+1
        } else {
            N=N/2
        }
        i=i+1
    }
    i
}
```

460 µs

- Bytecoded by Tcl
- No function call
- Dynamic data types

- Vectors used as scalars
- 4.5 function calls per iteration

VecTcl sucks at...

Complex operations

```
vexpr {
  r=x.*x + y.*y
}
```

⟷

```
for (int i=0; i<N; i++) {
  r[i] = x[i]*x[i] + y[i]*y[i];
}
```

- $2N$ Flops
- $2N$ temporary storage
- 3 passes over the data

- $2N$ Flops
- 2 temporary registers
- 1 pass over the data

```
vexpr {
  t1=x.*x
  t2=y.*y
  r=t1+t2
}
```

Compilation experiment

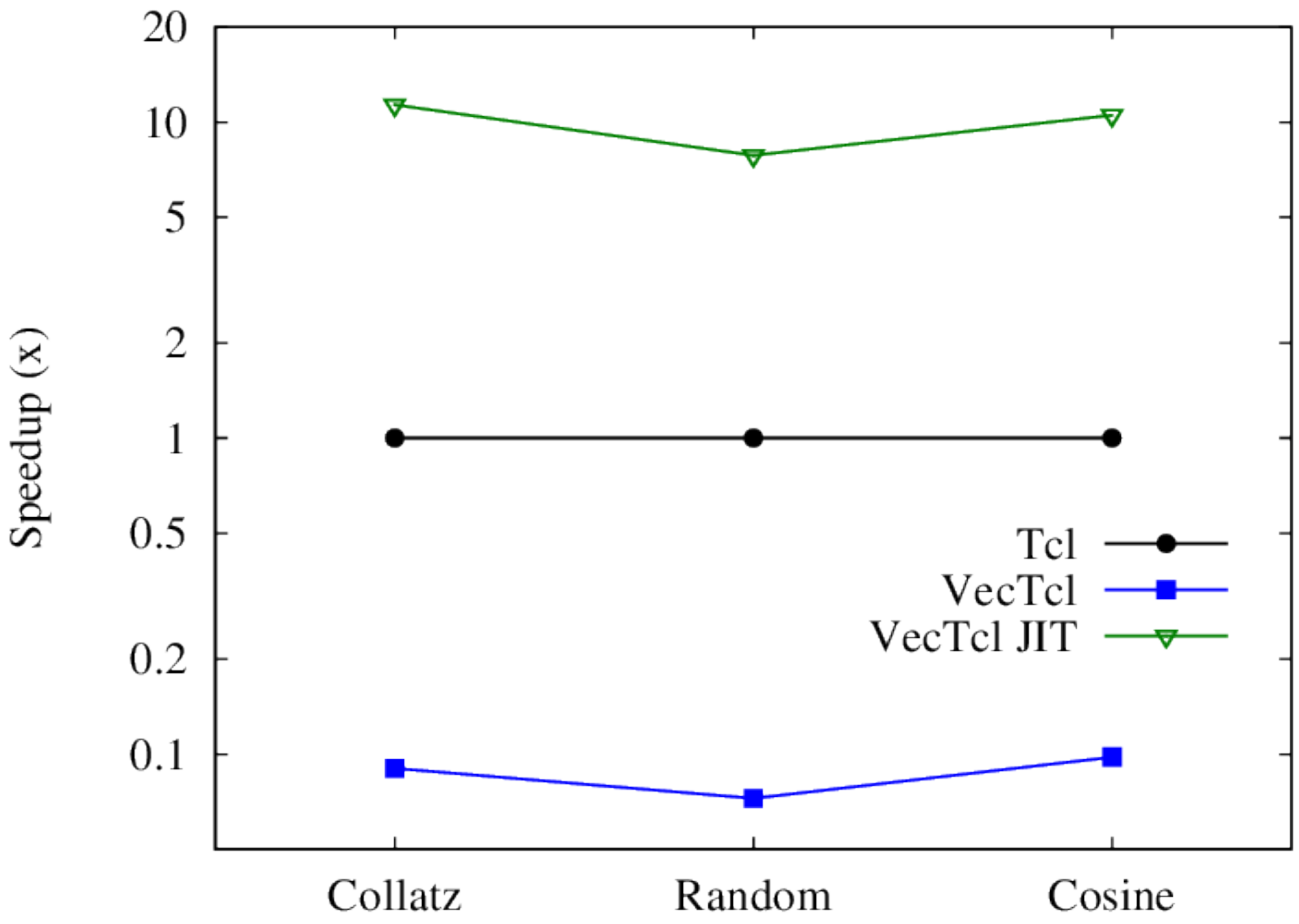
Both cases can be sped up by native compilation

```
vectcl::jitproc squares {{xv {double n}} {yv {double n}}}   {
     xv.*xv+yv.*yv
}
```

- Branch jit on github
- Code compiled to SSA, then C
- C code is compiled and linked using tcc4tcl
- Arguments are type-annotated

Scalar math result

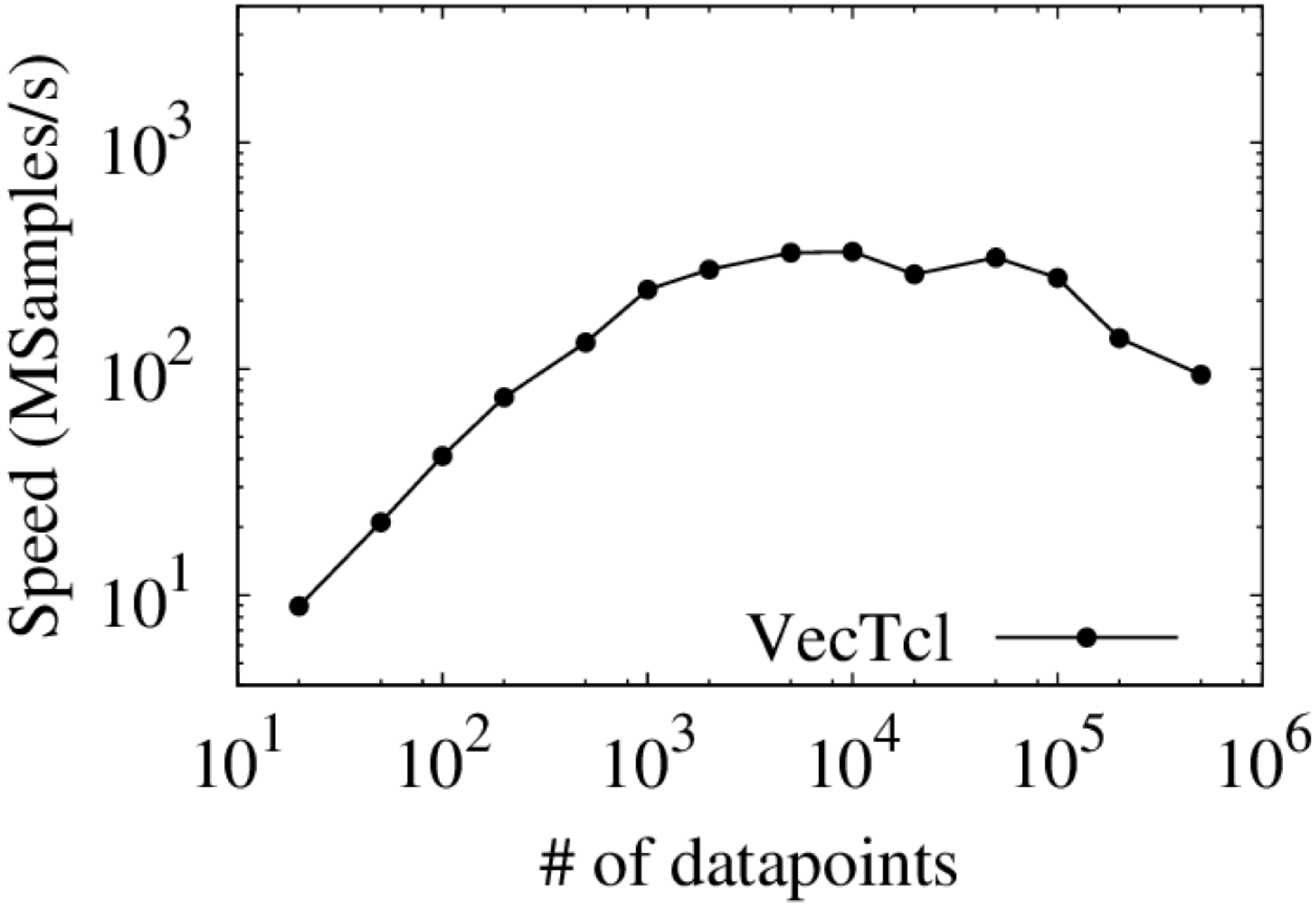
```
jitproc cos_jit {{x {double 1}}
{n {int 1}}} {
    j=0
    s=1.0
    t=1.0
    i=0
    while (i < n) {
        t=0-t*x*x / (j+1) / (j+2)
        s =s + t
        j=j+2
        i=i+1
    }
    s
}
```



- 10x increase over Tcl, 100x over pure VecTcl
- C code looks similar to handwritten code

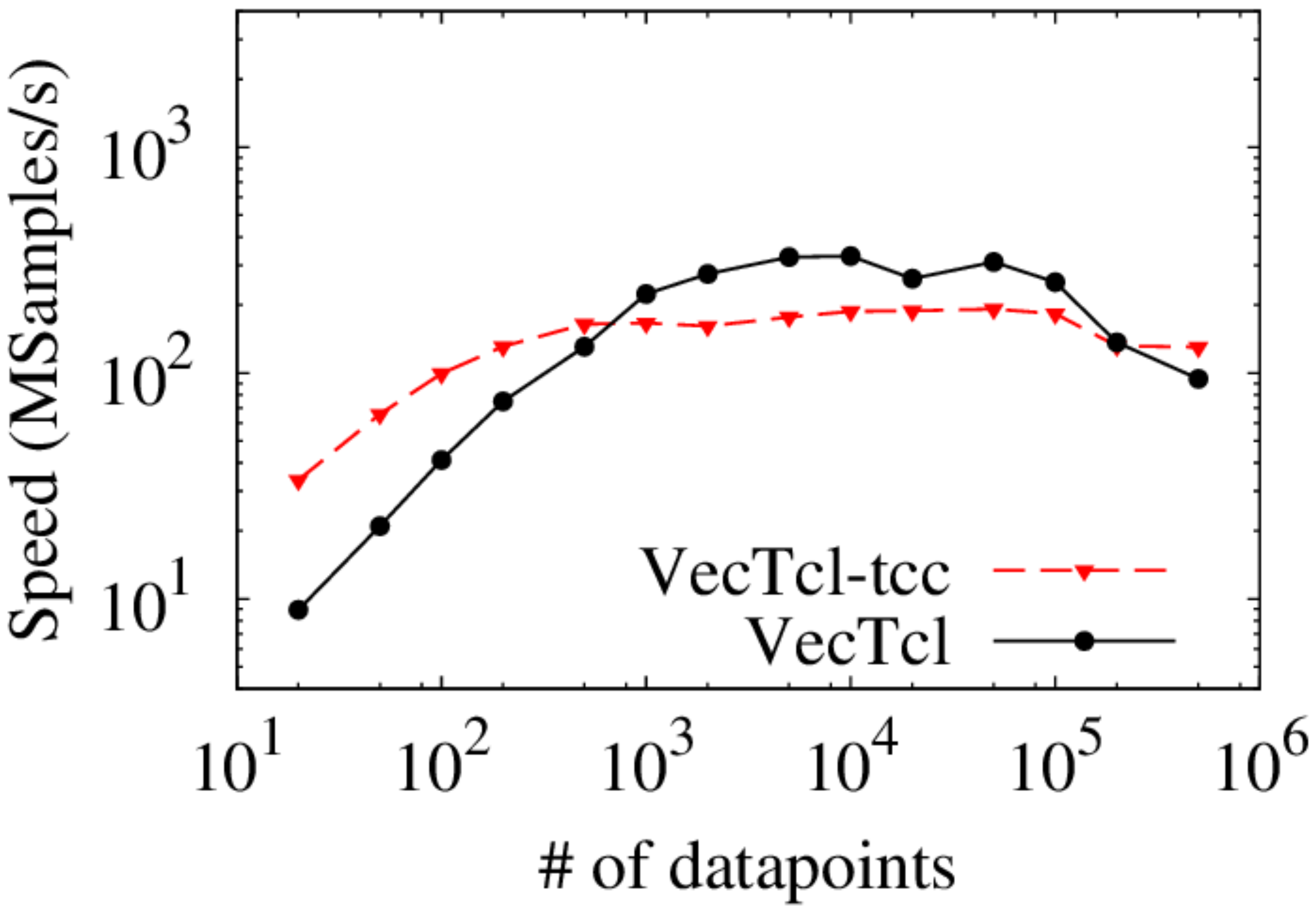
Squares result

```
vectcl::jitproc squares {{xv {double n}} {yv {double n}}}   {
      xv.*xv+yv.*yv
}
```



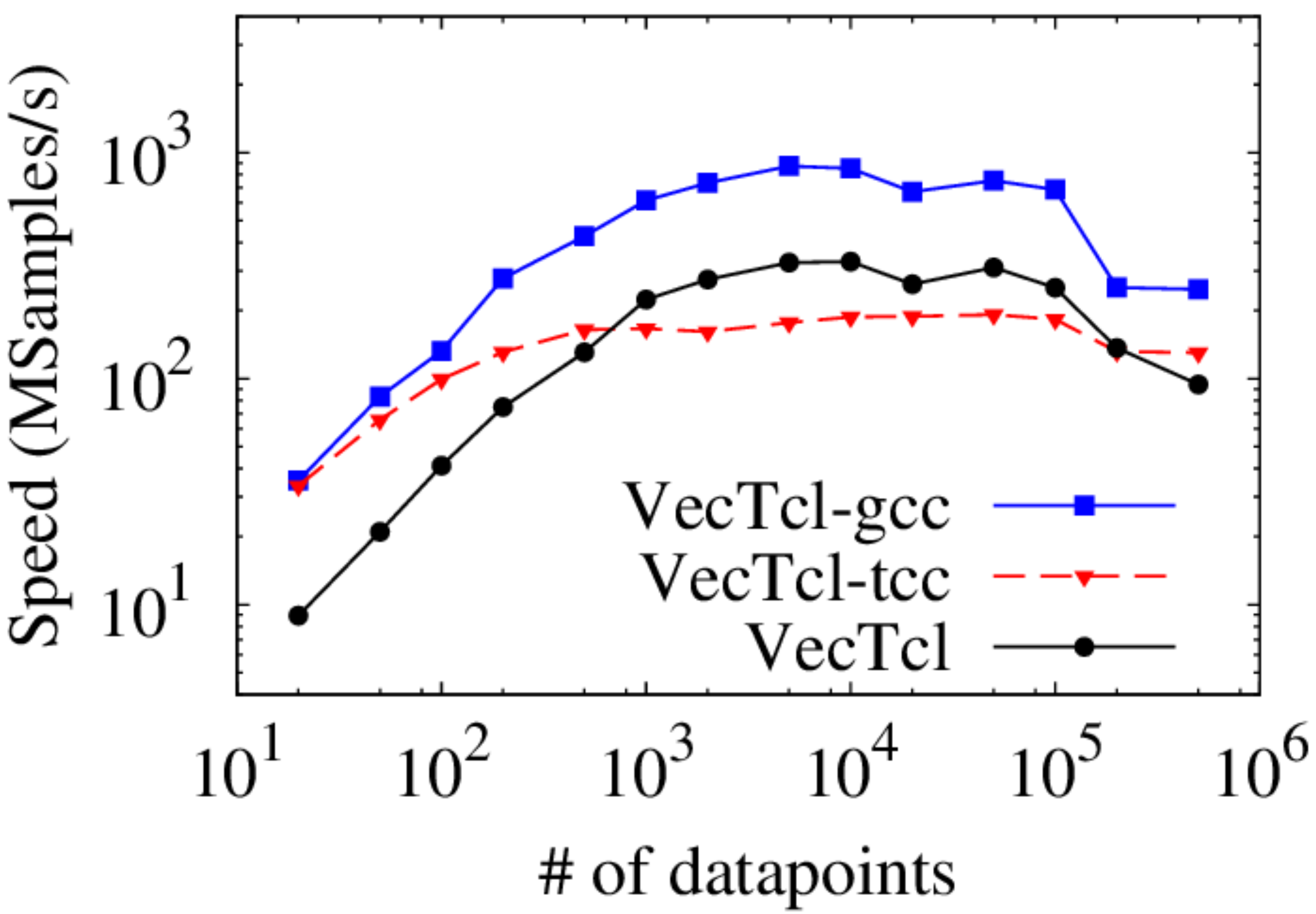
Squares result

```
vectcl::jitproc squares {{xv {double n}} {yv {double n}}}   {
      xv.*xv+yv.*yv
}
```



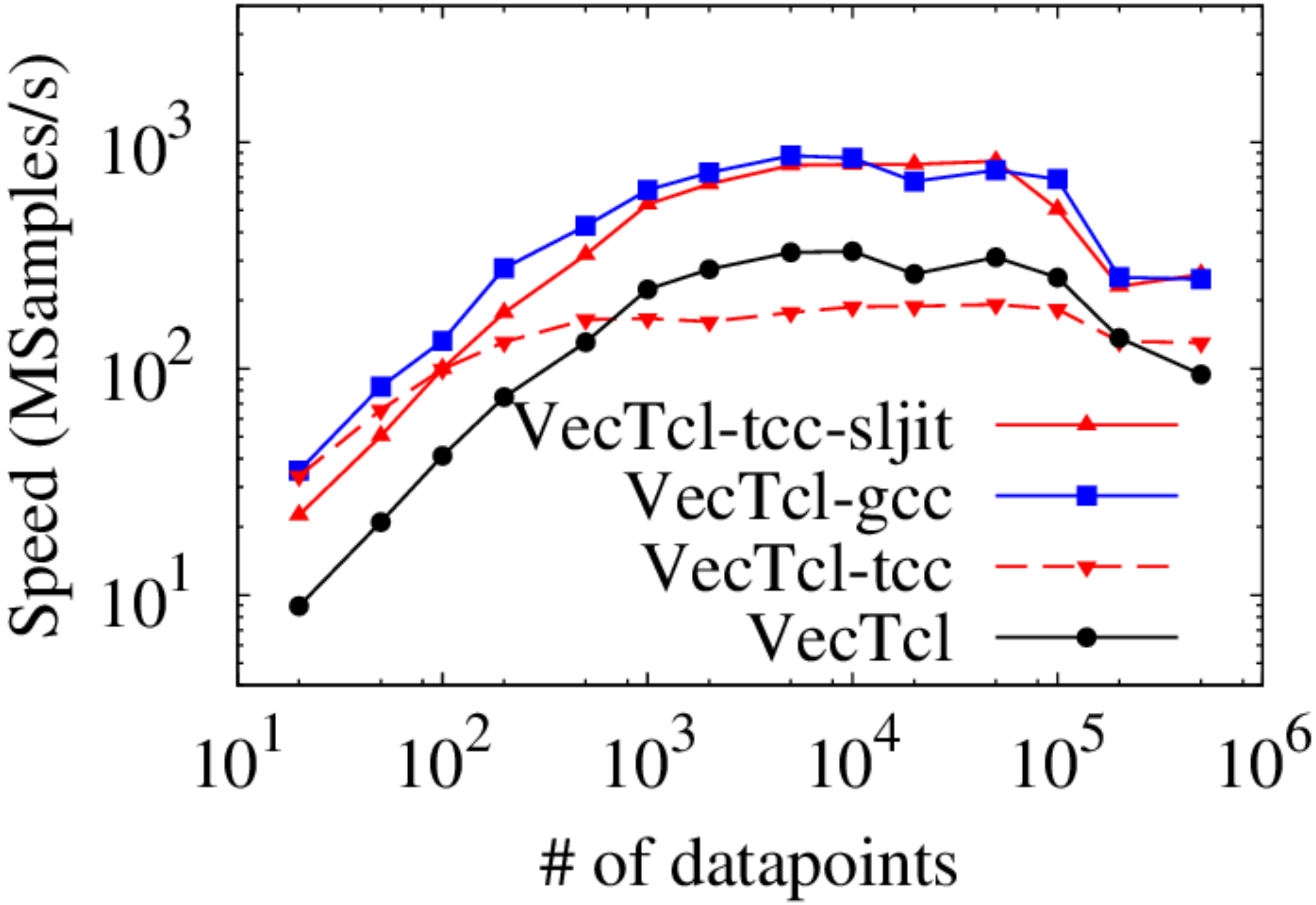
Squares result

```
vectcl::jitproc squares {{xv {double n}} {yv {double n}}}   {
      xv.*xv+yv.*yv
}
```



Squares result

```
vectcl::jitproc squares {{xv {double n}} {yv {double n}}}   {
      xv.*xv+yv.*yv
}
```



- JIT compiler cuts down one time cost
- tcc is too weak to beat standard VecTcl
- Inner loops could be compiled using a JIT library

Obstacles with compilation

Can I use it already?

Yes, you can, but....

... you don‘t want to!

Obstacles with compilation

Can I use it already?

Yes, you can, but....

- No slices
- No for loops (only while)
- Reductions aren‘t working properly
- Function calls mess up type inference
- Argument types must be given
- Certainly many bugs

... you don‘t want to!

Obstacles with compilation

Was it difficult to do?

Some things will never work / are impossible:

```
proc setx {v} {
    upvar 1 x x
    set x $v
}

vproc test {y} {
    setx(y)
    3*x
}
```

Obstacles with compilation

Was it difficult to do?

- 2000 LOT (lines of Tcl)

Some things will never work / are impossible:

- upvar, uplevel and traces

```
proc setx {v} {
    upvar 1 x x
    set x $v
}

vproc test {y} {
    setx(y)
    3*x
}
```

- Variable x doesn‘t even exist -> compiler error
- If it exists, it is a C local variable, inaccessible from outside

Obstacles with compilation

Was it difficult to do?

Some things will never work / are impossible:

```
proc mysurprise {i} {
    if {$i > 3} { return -code break }
    expr {$i*2}
}

vproc test {} {
    x=0
    for i=1:10 {
        x=x+mysurprise(i)
    }
    x
}
```

Obstacles with compilation

Was it difficult to do?

- 2000 LOT (lines of Tcl)

Some things will never work / are impossible:

- return codes

```
proc mysurprise {i} {
    if {$i > 3} { return -code break }
    expr {$i*2}
}

vproc test {} {
    x=0
    for i=1:10 {
        x=x+mysurprise(i)
    }
    x
}
```

Obstacles with compilation

Was it difficult to do?

Some things will never work / are impossible:

Hard to get right:

```
vproc test {} {
    x=0
    x+somefunc()
}
```

Obstacles with compilation

Was it difficult to do?

- 2000 LOT (lines of Tcl)

Some things will never work / are impossible:

- Dynamic code changes
- Redefinition of builtins (numarray::+ & friends)

Hard to get right:

- Function calls: What is the return type of test?

```
vproc test {} {
    x=0
    x+somefunc()
}
```

Obstacles with speed-up

tcc4tcl

- ⊕ Small footprint (~1MB)
- ⊕ ANSI C
- ⊕ Easy code generation (accepts C)
- ⊖ Weak optimizer (register allocation)

LLVM

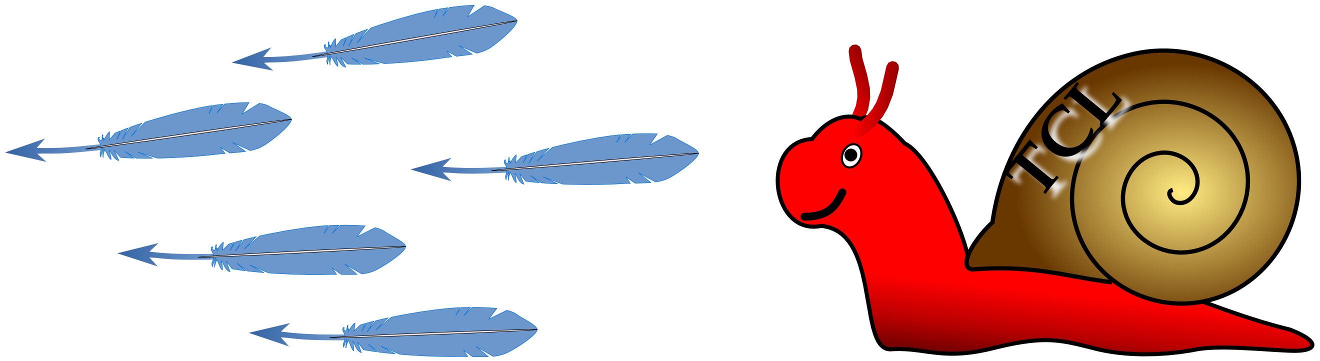
- ⊖ Large library
- ⊖ C++
- ⊖ Needs LLVM bytecode
- ⊕ Strong optimizer

sljit, NanoJIT, LuaJIT, ORC

- ⊖ Too limited, C++, restrictive license...

Conclusion & The Future

- VecTcl provides an easy interface to numeric math in Tcl
- Performance superior to other packages, but worse than C
- JIT compilation possible for restricted subset, speed-up 10× –100×
- tcc backend provides too weak optimization
- Rewrite in C++/LLVM ?

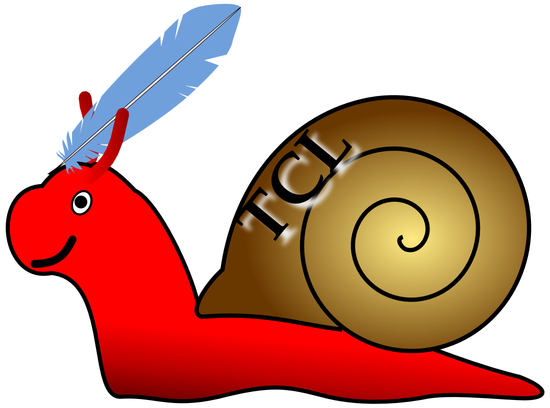


Why or when is it slow?

Tightly coded loops:

```
vexpr {
   a=zeros(1000);
   for i=0:999 {a[i]=2*i}
}
```

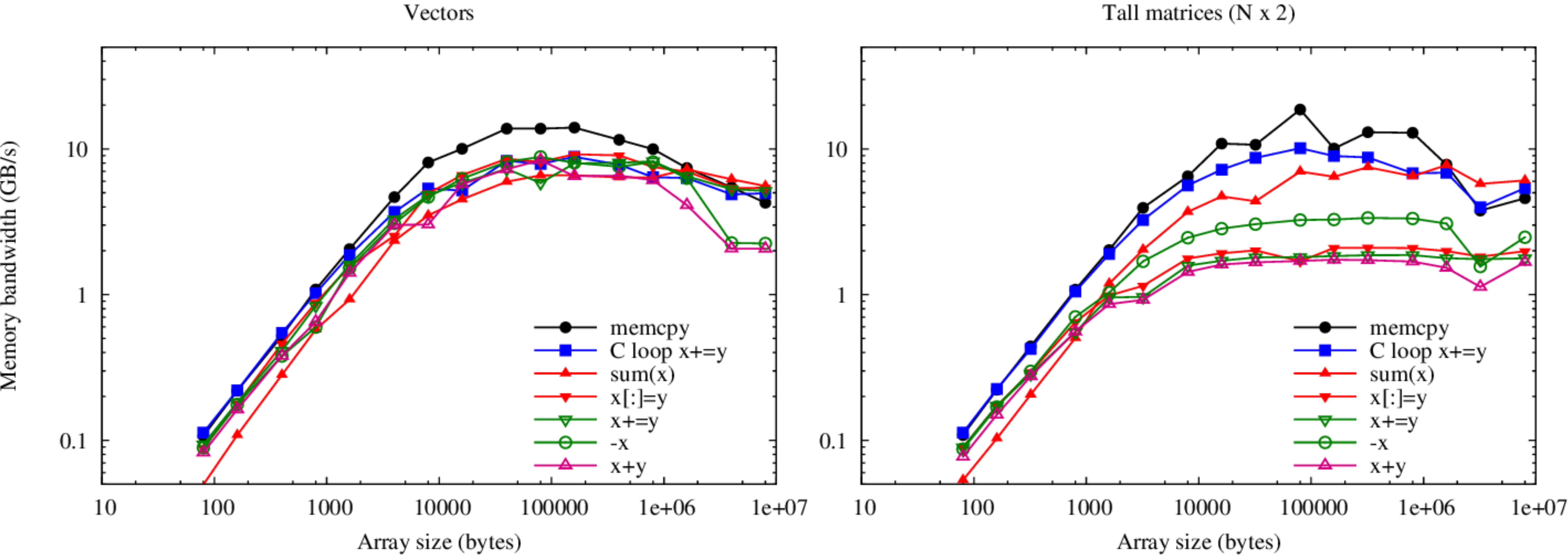
```
set a [zeros 1000]
set __temp1 999
for {set i 0} {$i <= $__temp1} {incr i 1} {
numarray::= a [list [list [set i] [set i] 1]]
[numarray::* 2 [set i]]
}
```



- Avoid if possible: `vexpr { a=2*linspace(0,999,1) }`
- Huge speed-up possible by JIT compilation (tcc4tcl?)

Why or when is it slow?

Speed of the elementary operations



- Vector operations close to the memory bandwidth
- Until ~10kbytes, the command dispatch dominates
- Matrix shape (currently) has a strong effect
- Improvement by OpenMP, BLAS, better iterators

No external dependencies

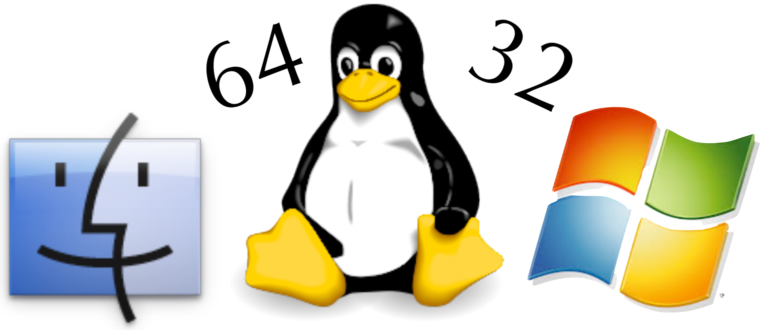
To compile VecTcl you need:

- a C compiler
- Tcl

FORTRAN
C++
GPL©

To run VecTcl you need:

- VecTcl
- TclOO



To rebuild VecTcl from scratch:

- autoconf
- tcllib::parsertools
- CLAPACK