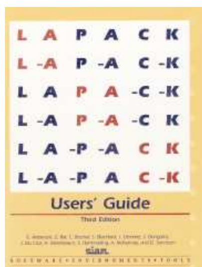
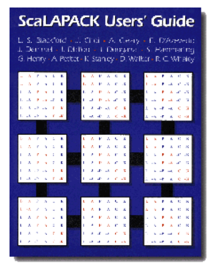
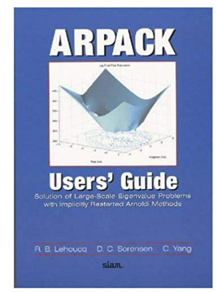
A Brief Feel for the Dream

In [ ]:
```
using Distributed
addprocs(4)
using LinearAlgebra, DistributedArrays, CuArrays
import CUDAdrv: @time
```

So you like linear algebra?







but you are innovating beyond the traditional libraries?

For demonstration let's make a matrix where you know more than the libraries know.

Let's create a custom matrix structure that depends on n parameters, not n^2 :

Diagonal(v) + v*v'

For example (n=3):

$$\begin{pmatrix} v_1 & & \\ & v_2 & \\ & & v_3 \end{pmatrix} + \begin{pmatrix} v_1^2 & v_1 v_2 & v_1 v_3 \\ v_2 v_1 & v_2^2 & v_2 v_3 \\ v_3 v_1 & v_3 v_2 & v_3^2 \end{pmatrix}.$$

Build a Custom Type

In [2]:
```
struct Sc19Matrix{T, V<:AbstractVector{T}} <: AbstractMatrix{T}
    v::V
end
```

In [3]:
```
Base.size(A::Sc19Matrix)           =   length(A.v), length(A.v)
Base.getindex(A::Sc19Matrix,i,j) =   A.v[i]*(i==j)  + A.v[i]*A.v[j]
```

In [4]:
```
A = Sc19Matrix([1,10,100])
```

Out[4]:
```
3×3 Sc19Matrix{Int64,Array{Int64,1}}:
   2    10    100
  10   110   1000
 100  1000  10100
```

In [5]:
```
dump( A )  # n storage
```

```
Sc19Matrix{Int64,Array{Int64,1}}
  v: Array{Int64}((3,)) [1, 10, 100]
```

In [6]:
```
dump(Matrix(A)) # n² storage
```

```
Array{Int64}((3, 3)) [2 10 100; 10 110 1000; 100 1000 10100]
```

My very own largest eigensolver for my very own matrices

In [7]:
```
f(A::Sc19Matrix)  = λ ->  1 + mapreduce((v) -> v^2 / (v - λ)  , +, A.v)
f′(A::Sc19Matrix) = λ ->      mapreduce((v) -> v^2 / (v - λ)^2, +, A.v)

function LinearAlgebra.eigmax(A::Sc19Matrix; tol = eps(2.0), debug = false)
    x0 = maximum(A.v) + maximum(A.v)^2
    δ = f(A)(x0)/f′(A)(x0)
    while abs(δ) > x0 * tol
        x0 -= δ
        δ = f(A)(x0)/f′(A)(x0)
        debug && println("x = $x0, δ = $δ") # Debugging
    end
    x0
end
```

In [8]:
```
eigmax(A)
```

Out[8]:
```
10200.107083707298
```

In [9]:
```
eigmax(Matrix(A))
```

Out[9]:
```
10200.107083707298
```

Go Heterogeneous

In [11]:
```
gpuA = Sc19Matrix(CuArray([1,2,3]))
```

Out[11]:
```
3×3 Sc19Matrix{Int64,CuArray{Int64,1}}:
 2  2   3
 2  6   6
 3  6  12
```

In [12]:
```
distA = Sc19Matrix(distribute([1,2,3]))
```

Out[12]:
```
3×3 Sc19Matrix{Int64,DArray{Int64,1,Array{Int64,1}}}:
 2  2   3
 2  6   6
 3  6  12
```

In [13]:
```
Matrix(gpuA)
```
Out[13]:
```
3×3 Array{Int64,2}:
 2  2   3
 2  6   6
 3  6  12
```

In [14]:
```
Matrix(distA)
```
Out[14]:
```
3×3 Array{Int64,2}:
 2  2   3
 2  6   6
 3  6  12
```

Compare Timings

In [15]:
```
N = 4_000_000
```
Out[15]:
```
4000000
```

In [16]:
```
v = randn(N)*.1
```
Out[16]:
```
4000000-element Array{Float64,1}:
  0.1272694901662584
  0.05590256637096559
 -0.05542613993142005
  0.02538511395183468
  0.03991025920667298
  0.09145530767524132
 -0.0008358771152277827
 -0.05838237707993508
 -0.19546358714068626
 -0.16993291596813465
 -0.08947655748828592
 -0.182111781531594
 -0.12210587515032732
  ⋮
  0.07867400049366316
 -0.014186684393361099
 -0.021207147327113367
 -0.1452749444007715
 -0.003807279596145612
 -0.07014815796070485
  0.15353822234976333
  0.03433408436873784
 -0.10435583020611175
  0.047915513621000404
 -0.0017681107131386103
 -0.09099331505222581
```

In [17]:
```
A = Sc19Matrix(v)
```

Out[17]:
```
4000000×4000000 Sc19Matrix{Float64,Array{Float64,1}}:
  0.143467      0.00711469   -0.00705406   …  -0.000225027  -0.0115807
  0.00711469    0.0590277    -0.00309846      -9.88419e-5   -0.00508676
 -0.00705406   -0.00309846   -0.0523541        9.79996e-5    0.00504341
  0.00323075    0.00141909   -0.001407        -4.48837e-5   -0.00230988
  0.00507936    0.00223109   -0.00221207      -7.05658e-5   -0.00363157
  0.0116395     0.00511259   -0.00506901   …  -0.000161703  -0.00832182
 -0.000106382  -4.67277e-5    4.63294e-5       1.47792e-6    7.60592e-5
 -0.0074303    -0.00326372    0.00323591       0.000103227   0.00531241
 -0.0248766    -0.0109269     0.0108338        0.000345601   0.0177859
 -0.0216273    -0.00949969    0.00941873       0.00030046    0.0154628
 -0.0113876    -0.00500197    0.00495934   …   0.000158204   0.00814177
 -0.0231773    -0.0101805     0.0100938        0.000321994   0.016571
 -0.0155404    -0.00682603    0.00676786       0.000215897   0.0111108
  ⋮                                        ⋱
  0.0100128     0.00439808   -0.0043606       -0.000139104  -0.00715881
 -0.00180553   -0.000793072   0.000786313      2.50836e-5    0.00129089
 -0.00269902   -0.00118553    0.00117543   …   3.74966e-5    0.00192971
 -0.0184891    -0.00812124    0.00805203       0.000256862   0.013219
 -0.000484551  -0.000212837   0.000211023      6.73169e-6    0.000346437
 -0.00892772   -0.00392146    0.00388804       0.00012403    0.00638301
  0.0195407     0.00858318   -0.00851003      -0.000271473  -0.013971
  0.00436968    0.00191936   -0.00190301   …  -6.07065e-5   -0.00312417
 -0.0132813    -0.00583376    0.00578404       0.000184513   0.00949568
  0.00609818    0.0026786    -0.00265577      -8.47199e-5   -0.00435999
 -0.000225027  -9.88419e-5    9.79996e-5      -0.00176498    0.000160886
 -0.0115807    -0.00508676    0.00504341       0.000160886  -0.0827135
```

In [18]:
```
distA = Sc19Matrix(distribute(v));
```

In [19]:
```
gpuA = Sc19Matrix(CuArray(v));
```

In [21]:
```
@time eigmax(A) # run twice
```

```
  0.442624 seconds (5 allocations: 176 bytes)
```

Out[21]:
```
39995.34634455378
```

In [23]:
```
@time eigmax(distA) # run twice
```

```
  0.173184 seconds (22.30 k allocations: 859.672 KiB, 3.85% gc time)
```

Out[23]:
```
39995.34634455378
```

In [25]:
```
@time eigmax(gpuA) # run twice
```

```
  0.008704 seconds (4.51 k allocations: 163.188 KiB)
```

Out[25]:
```
39995.34634455378
```

Abstraction

1. A data structure is a mathematical matrix!
2. A matrix can be serial, distributed, gpu
   BOTH ARE ABSTRACTIONS, BOTH use underlying similar mechanisms

Why Julia?

1. Well designed abstractions
2. Multiple dispatch
3. Careful balance between static and dynamic
4. Compiles to GPU at multiple levels, not just lowest CUDA level
5. Metaprogramming across the stack
6. Interfaces with LLVM
7. Plays nicely with Python, legacy codes, ...

[Julia: A Fresh Approach to Numerical Computing (https://arxiv.org/abs/1411.1607)](https://arxiv.org/abs/1411.1607)

SIAM Rev., 59(1), 65–98. (34 pages)

Julia: A Fresh Approach to Numerical Computing

Jeff Bezanson, **Alan Edelman**, **Stefan Karpinski**, **and Viral B. Shah**

https://doi.org/10.1137/141000671

Bridging cultures that have often been distant, Julia combines expertise from the diverse fields of computer science and computational science to create a new approach to numerical computing. Julia is designed to be easy and fast and questions notions generally held to be “laws of nature" by practitioners of numerical computing: \beginlist \item High-level dynamic programs have to be slow. \item One must prototype in one language and then rewrite in another language for speed or deployment. \item There are parts of a system appropriate for the programmer, and other parts that are best left untouched as they have been built by the experts. \endlist We introduce the Julia programming language and its design---a dance between specialization and abstraction. Specialization allows for custom treatment. *Multiple dispatch*, a technique from computer science, picks the right algorithm for the right circumstance. Abstraction, which is what good computation is really about, recognizes what remains the same after differences are stripped away. Abstractions in mathematics are captured as code through another technique from computer science, *generic programming*. Julia shows that one can achieve machine performance without sacrificing human convenience.

In [ ]: