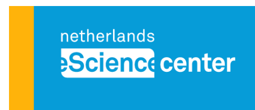
GalerkinToolkit.jl:
Towards a multi-level Finite Element API for CPUs and GPUs

Francesc Verdugo (and collaborators)

GalerkinToolkit

High-performance
finite element toolbox
in Julia



Grant ID: NLESC.SS.2023.008

Julia4PDEs workshop · 2026-03-26

The team

Main developer

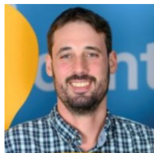
- Francesc Verdugo (VU)

GalerkinToolkit form compiler

- Xiaowei Ouyang (VU)
- Tiziano de Matteis (VU)

Software sustainability & GPU

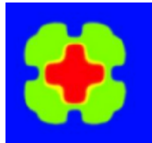
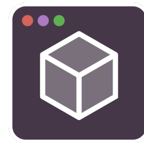
- Robin Richardson (NLeSc)
- Stijn Heldens (NLeSc)
- Alessio Sclocco (NLeSc)

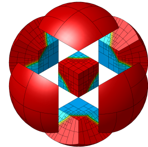


Why another FEM library?

VU

- Deal.ii
- GalerkinToolkit.jl
- Gridap.jl
- FEniCS
- Ferrite.jl
- Firedrake
- MFEM
- ...



Agenda

This talk

- Multi-level API
- What is new in our code generation strategy?
- Towards GPU support

Next: Xiaowei's talk

- The GalerkinToolkit form compiler (GTFC)

Next: GalerkinToolkit demo

Simple FEM in math notation

Let $f, g : \mathbb{R}^d \longrightarrow \mathbb{R}$ be two functions.
Let $\Omega \subset \mathbb{R}^d$ be a computational domain.
Let $\mathcal{T}$ be a *triangulation* of Ω
Let $V := \{v \in H^1(\Omega) : \ v(x) \text{ is a polynomial for all } x \in T \in \mathcal{T}\}$
Let $V_0 := \{v \in V : \ v(x) = 0 \text{ for } x \in \partial\Omega\}$
Let $V_g := \{v \in V : \ v(x) = g(x) \text{ for x} \in \partial\Omega\}$

Find $u \in V_g$ such that:
$\int_\Omega \nabla u(x) \cdot \nabla v(x) \, \mathrm{dx} = \int_\Omega f(x) v(x) \, \mathrm{dx}$ for all $v \in V_0$

"High-Level" API

```julia
1 using LinearAlgebra
2 import Gmsh
3 import GalerkinToolkit as GT
4 import LinearSolve
5 import ForwardDiff
6 import GLMakie
7 mesh = GT.mesh_from_msh("model.msh")
8 Ω = GT.interior(mesh)
9 Γd = GT.boundary(mesh)
10 f = GT.analytical_field(x->1.0,Ω)
11 g = GT.analytical_field(x->2.0,Ω)
12 k = 1
13 V = GT.lagrange_space(Ω,k;dirichlet_boundary=Γd)
```

```julia
14 T = Float64
15 uhd = GT.zero_dirichlet_field(T,V)
16 GT.interpolate_dirichlet!(g,uhd)
17 degree = 2*k
18 dΩ = GT.measure(Ω,degree)
19 ∇ = ForwardDiff.gradient
20 a = (u,v) -> GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
21 l = v -> GT.∫( x->v(x)*f(x), dΩ)
22 p = GT.SciMLBase_LinearProblem(uhd,a,l)
23 sol = LinearSolve.solve(p)
24 uh = GT.solution_field(uhd,sol)
25 GT.makie_surfaces(Ω;color=uh)
```


```julia
20 a = (u,v) -> GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
21 l = v -> GT.∫( x->v(x)*f(x), dΩ)
22 p = GT.SciMLBase_LinearProblem(uhd,a,l)
```

”Low-level” API

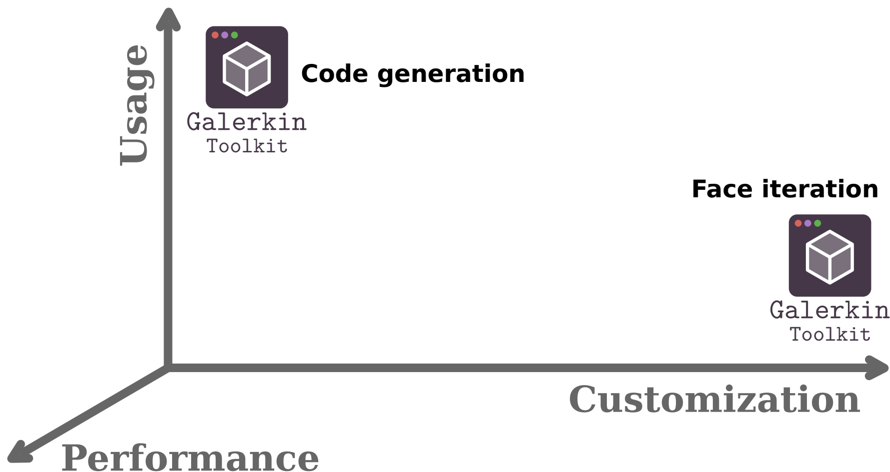
```julia
1 using LinearAlgebra
2 import Gmsh
3 import GalerkinToolkit as GT
4 import LinearSolve
5 import ForwardDiff
6 import GLMakie as Makie
7 mesh = GT.mesh_from_msh("model.msh")
8 Ω = GT.interior(mesh)
9 Γd = GT.boundary(mesh)
10 f = GT.analytical_field(x->1.0,Ω)
11 g = GT.analytical_field(x->2.0,Ω)
12 k = 1
13 V = GT.lagrange_space(Ω,k;dirichlet_boundary=Γd)
14 T = Float64
15 uhd = GT.zero_dirichlet_field(T,V)
16 GT.interpolate_dirichlet!(g,uhd)
17 degree = 2*k
18 dΩ = GT.measure(Ω,degree)
19 A_alloc = GT.allocate_matrix(T,V,V,Ω)
20 free_or_dirichlet=(GT.FREE,GT.DIRICHLET)
21 Ad_alloc = GT.allocate_matrix(T,V,V,Ω;free_or_dirichlet)
22 b_alloc = GT.allocate_vector(T,V,Ω,Γn)
23 function assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
24     tabulate = (GT.value,∇)
25     compute = (GT.coordinate,)
26     V_faces = GT.each_face(V,dΩ;tabulate,compute)
27     n = GT.max_num_reference_dofs(V)
28     T = Float64
29     Auu = zeros(T,n,n)
30     bu = zeros(T,n)
31     for V_face in V_faces
32         dofs = GT.dofs(V_face)
33         fill!(Auu,zero(T))
34         fill!(bu,zero(T))
35         for V_point in GT.each_point(V_face)
36             x = GT.coordinate(V_point)
37             dV = GT.weight(V_point)
38             dof_s = GT.shape_functions(GT.value,V_point)
39             dof_∇s = GT.shape_functions(∇,V_point)
40             for (i,dofi) in enumerate(dofs)
41                 v = dof_s[i]
42                 ∇v = dof_∇s[i]
43                 bu[i] += f.definition(x)*v*dV
44                 for (j,dofj) in enumerate(dofs)
45                     ∇u = dof_∇s[j]
46                     Auu[i,j] += ∇v⋅∇u*dV
47                 end
48             end
49         end
50         GT.contribute!(A_alloc,Auu,dofs,dofs)
51         GT.contribute!(Ad_alloc,Auu,dofs,dofs)
52         GT.contribute!(b_alloc,bu,dofs)
53     end
54 end
55 assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
56 A = GT.compress(A_alloc)
57 Ad = GT.compress(Ad_alloc)
58 b = GT.compress(b_alloc)
59 xd = GT.dirichlet_values(uhd)
60 b .= b .- Ad*xd
61 p = LinearSolve.LinearProblem(A,b)
62 sol = LinearSolve.solve(p)
63 uh = GT.solution_field(uhd,sol)
64 GT.makie_surfaces(Ω;color=uh)
```

”Low-level” API

```julia
23 function assemble_in_Ω!(A_alloc,Ad_alloc,b_alloc,V,f,dΩ)
24     tabulate = (GT.value,∇)
25     compute = (GT.coordinate,)
26     V_faces = GT.each_face(V,dΩ;tabulate,compute)
27     n = GT.max_num_reference_dofs(V)
28     T = Float64
29     Auu = zeros(T,n,n)
30     bu = zeros(T,n)
31     for V_face in V_faces
32         dofs = GT.dofs(V_face)
33         fill!(Auu,zero(T))
34         fill!(bu,zero(T))
35         for V_point in GT.each_point(V_face)
36             x = GT.coordinate(V_point)
37             dV = GT.weight(V_point)
38             dof_s = GT.shape_functions(GT.value,V_point)
39             dof_∇s = GT.shape_functions(∇,V_point)
40             for (i,dofi) in enumerate(dofs)
41                 v = dof_s[i]
42                 ∇v = dof_∇s[i]
43                 bu[i] += f.definition(x)*v*dV
44                 for (j,dofj) in enumerate(dofs)
45                     ∇u = dof_∇s[j]
46                     Auu[i,j] += ∇v⋅∇u*dV
47                 end
48             end
49         end
50         GT.contribute!(A_alloc,Auu,dofs,dofs)
51         GT.contribute!(Ad_alloc,Auu,dofs,dofs)
52         GT.contribute!(b_alloc,bu,dofs)
53     end
54 end
```

```julia
26     V_faces = GT.each_face(V,dΩ;tabulate,compute)
35         for V_point in GT.each_point(V_face)
```

API trade-offs



GalerkinToolkit form compiler (GTFC)

```
GT.∫( x->∇(u,x)⋅∇(v,x), dΩ)
```

```
function assemble_in_Ω!(A_alloc,V,dΩ)
    n = GT.max_num_reference_dofs(V)
    Auu = zeros(Float64,n,n)
    for V_face in GT.each_face(V,dΩ;tabulate=(∇,))
        dofs = GT.dofs(V_face)
        fill!(Auu,zero(Float64))
        for V_point in GT.each_point(V_face)
            dV = GT.weight(V_point)
            dof_∇s = GT.shape_functions(∇,V_point)
            for (i,dofi) in enumerate(dofs)
                ∇v = dof_∇s[i]
                for (j,dofj) in enumerate(dofs)
                    ∇u = dof_∇s[j]
                    Auu[i,j] += ∇v⋅∇u*dV
                end
            end
        end
        GT.contribute!(A_alloc,Auu,dofs,dofs)
    end
end
```

* The actual generated code is slightly different

User-defined functions and types

Let $\sigma : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ be a user-defined flux

The weak form transforms into

$$\int_\Omega \sigma(\nabla u(x)) \cdot \nabla v(x) \, \mathrm{d}x = \int_\Omega v(x) \, \mathrm{d}x$$

```
function σ(∇u)
    # You can implement here
    # anything that
    # returns a vector of
    # the same length as ∇u
end
```

```
GT.∫(dΩ) do x
    σ_u = GT.external(σ,∇(u,x))
    σ_u⋅∇(v,x)
end
```

User-defined functions and types

```
function assemble_in_Ω!(A_alloc,V,dΩ)
    n = GT.max_num_reference_dofs(V)
    Auu = zeros(Float64,n,n)
    σ_u = # allocate σ_u
    for V_face in GT.each_face(V,dΩ;tabulate=(∇,))
        dofs = GT.dofs(V_face)
        fill!(Auu,zero(Float64))
        for V_point in GT.each_point(V_face)
            dV = GT.weight(V_point)
            dof_∇s = GT.shape_functions(∇,V_point)
            for (j,dofj) in enumerate(dofs)
                ∇u = dof_∇s[j]
                σ_u[j] = σ(∇u)
            end
            for (i,dofi) in enumerate(dofs)
                ∇v = dof_∇s[i]
                for (j,dofj) in enumerate(dofs)
                    Auu[i,j] += σ_u[j]⋅∇u*dV
                end
            end
        end
        GT.contribute!(A_alloc,Auu,dofs,dofs)
    end
end
```


```
dof_∇s = GT.shape_functions(∇,V_point)
for (j,dofj) in enumerate(dofs)
    ∇u = dof_∇s[j]
    σ_u[j] = σ(∇u)
end
for (i,dofi) in enumerate(dofs)
    ∇v = dof_∇s[i]
    for (j,dofj) in enumerate(dofs)
        Auu[i,j] += σ_u[j]⋅∇u*dV
    end
end
```


* The actual generated code is slightly different

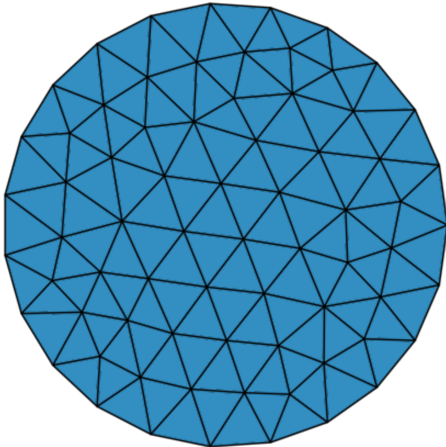
Example: Interfacing with Tensors.jl

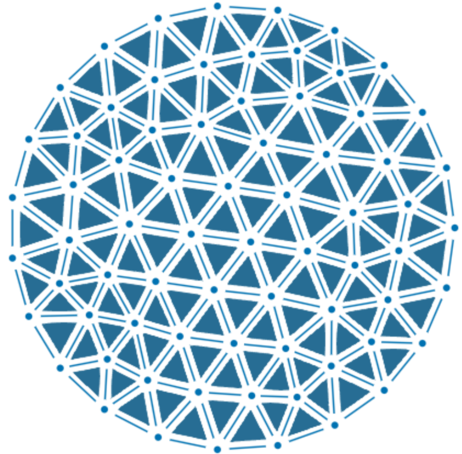
```julia
#Definition of strain
function ε(∇u)
    ∇u_tensor =Tensors.Tensor{2,3}(∇u)
    Tensors.symmetric(∇u_tensor)
end

#Definition of stress from strain
function σ(ε)
    λ*tr(ε)*one(ε) + 2*μ*ε
end
```

```julia
#Bi-linear form
function a(u,v)
    GT.∫(dΩ) do x
        ∇_v = GT.jacobian(v,x)
        ∇_u = GT.jacobian(u,x)
        ε_v = GT.external(ε,∇_v)
        ε_u = GT.external(ε,∇_u)
        σ_u = GT.external(σ,ε_u)
        GT.external(Tensors.dcontract,σ_u,ε_v)
    end
end
```

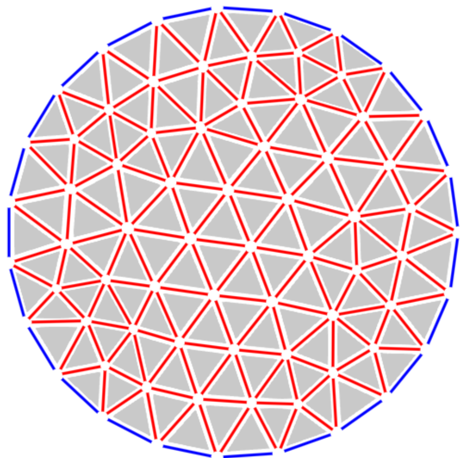
Polyhedral complexes

VU





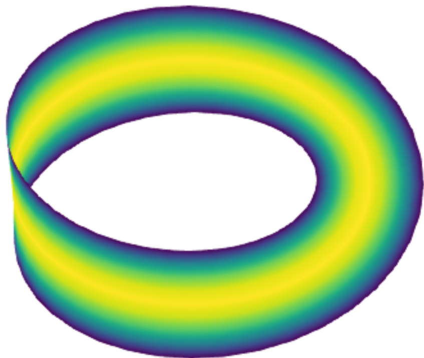
Polyhedral complexes

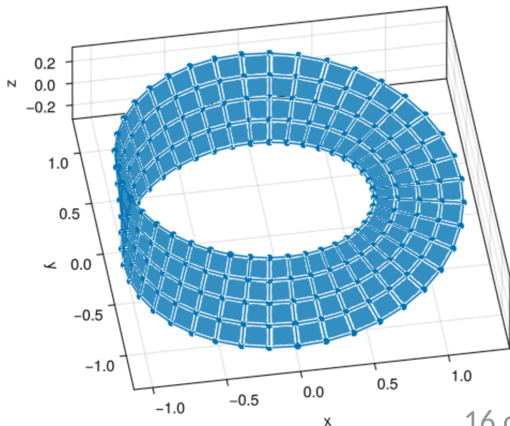


```
GT.∫(x->v(x),dΩ)

GT.∫(x->v(x),dΓ)

GT.∫(x->v[1](x)-v[2](x),dΛ)

GT.∫(x->v(x)+q(x),dΓ)
```

Manifolds

Example: Laplace-Beltrami on a Möbius strip





Parametrizable code generation

Example: Residual of a non-linear problem (p-Laplacian).

$r(v) = \int_\Omega \nabla v \cdot \left(|\nabla u_h|^{p-2} \, \nabla u_h \right) \, \mathrm{d}\Omega$

```
uh = # Define your FE field
flux(∇u) = norm(∇u)^(p-2) * ∇u
r = v -> GT.∫( x-> ∇(v,x)⋅GT.external(flux,∇(uh,x)), dΩ)
# Assembly (code gets generated here)
b,cache = GT.assemble_vector(r,Float64,V;parameters=(uh,))


uh2 = # Another FE field
# Reuse the code generated above
GT.update_vector!(b,cache;parameters=(uh2,))
```

Building SciML problems

```
# ODE Problem
m = (u,v) -> GT.∫(x->C*v(x)*u(x),dΩ)
r = (uh,t) -> v -> -1*GT.∫(x->∇(uh,x)⋅∇(v,x),dΩ)
j = (uh,t) -> (u,v) -> -1*GT.∫(x->∇(u,x)⋅∇(v,x),dΩ)
tspan = (0.0,T)
problem = GT.SciMLBase_ODEProblem(tspan,uh,m,r,j;dirichlet_dynamics!)
integrator = DifferentialEquations.init(problem)
for integrator in integrator
    uh = GT.solution_field(integrator)
end
```

Performance: Gridap vs GalerkinToolkit

gridap gt

Poisson equation, continuous Galerkin and Nitsche

Poisson equation, Discontinuous Galerkin

Stokes equation

Poisson equation, continuous Galerkin

Cells/second

Interpolation order (1, 2, or 3)

Higher is better. Results for unstructured hexahedal meshes. Internal git branch.

Performance: FEniCSx vs GalerkinToolkit

fenicsx gt

Poisson equation, continuous Galerkin and Nitsche

Poisson equation, Discontinuous Galerkin

Stokes equation

Poisson equation, continuous Galerkin

Cells/second

Interpolation order (1, 2, or 3)

Higher is better. Results for unstructured hexahedal meshes. Internal git branch.

Porting low-level API to GPUs

Example: Integrate a user-defined function on the CPU

$$\int_\Omega f(x)\,\mathrm{d}x$$

```
Ω = # define domain here
f(x) = 2*sin(sum(x))

degree = 5
dΩ = GT.quadrature(Ω,degree)
s = 0.0
for dΩ_face in GT.each_face(dΩ)
    for dΩ_point in GT.each_point(dΩ_face)
        x = GT.coordinate(dΩ_point)
        dx = GT.weight(dΩ_point)
        s += f(x)*dx
    end
end
```

Porting low-level API to GPUs

Example: Integrate a user-defined function on the GPU

```
function kernel!(contributions,dΩ_faces_gpu)
    face_id = (blockIdx().x-1)*blockDim().x +
              threadIdx().x
    if face_id > length(dΩ_faces_gpu)
        return nothing
    end
    dΩ_face = dΩ_faces_gpu[face_id]
    s = 0.0
    for dΩ_point in GT.each_point(dΩ_face)
        x = GT.coordinate(dΩ_point)
        dx = GT.weight(dΩ_point)
        s += f(x)*dx
    end
    contributions[face_id] = s
    return nothing
end
```

```
dΩ_faces_cpu = GT.each_face(dΩ)
dΩ_faces_gpu = CUDA.cu(dΩ_faces_cpu)
nfaces = length(dΩ_faces_gpu)
contributions = CUDA.zeros(Float64,nfaces)
threads_in_block = 256
blocks_in_grid = ceil(Int, nfaces/256)
@cuda threads=threads_in_block
      blocks=blocks_in_grid
      kernel!(contributions,dΩ_faces_gpu)
s = sum(contributions)
```

Internal git branch.

Porting low-level API to GPUs

Example: Matrix-free Laplace matrix-vector product

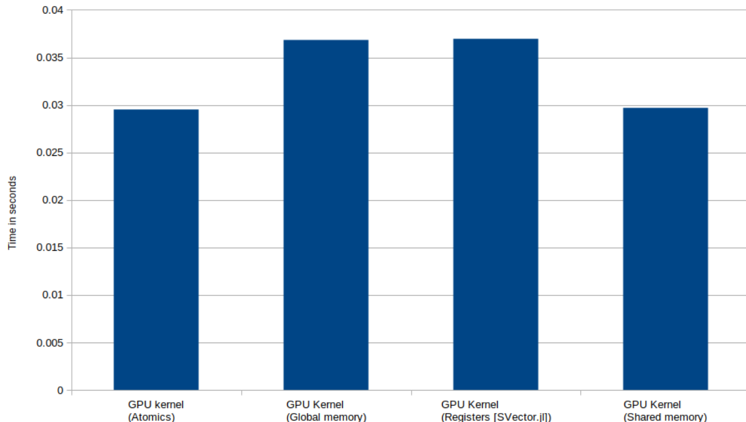
```
# b = A*x

# Discrete field on CPU
uh = GT.solution_field(V,x)

# Face iterator on CPU
tabulate = (GT.gradient,)
uh_faces = GT.each_face_new(uh,dΩ;tabulate)

# Face iterator on GPU
uh_faces_gpu = Adapt.adapt(dev,uh_faces)

# Result vector on GPU
ndofs = GT.num_free_dofs(V)
b_gpu = CUDA.zeros(Float64,ndofs)

# Kernel launch
threads_in_block = 256
blocks_in_grid = ceil(Int, nfaces/256)
@cuda threads=threads_in_block
      blocks=blocks_in_grid
      kernel!(b_gpu,uh_faces_gpu)
```

```
function kernel!(b,uh_faces)
    face_id = (blockIdx().x - 1) * blockDim().x +
        threadIdx().x
    if face_id > length(uh_faces)
        return nothing
    end
    uh_face = uh_faces[face_id]
    dofs = GT.dofs(uh_face)
    n = GT.num_dofs(uh_face)
    for uh_point in GT.each_point_new(uh_face)
        ux = GT.field(GT.gradient,uh_point)
        sx = GT.shape_functions(GT.gradient,uh_point)
        dx = GT.weight(uh_point)
        ux_dx = ux*dx
        for i in 1:n
            Atomix.@atomic b[dofs[i]] += ux_dx⋅sx[i]
        end
    end
    return nothing
end
```

Internal git branch.

Porting low-level API to GPUs

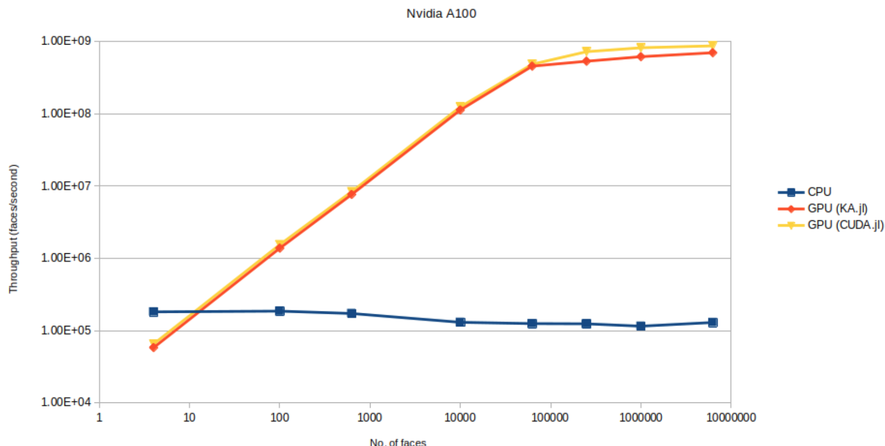
Example: Matrix-free Laplace matrix-vector product



Lower is better. Internal git branch.

Porting low-level API to GPUs

Example: Matrix-free Laplace matrix-vector product



Higher is better. Internal git branch.

GalerkinToolkit: Summary

- New multi-purpose FEM package in Julia
- Multi-level API
- Inter-operable with Julia package ecosystem
- Towards GPU support
- It is FOSS!