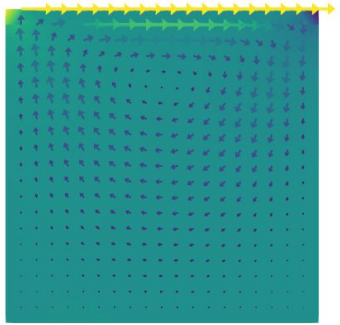
# Expressive and efficient Finite Element code generation in Julia: the progress of GalerkinToolkit form compiler



**Xiaowei Ouyang**
Tiziano De Matteis
Francesc Verdugo

GalerkinToolkit

# Background: Why code generation?

- There is no solution for all. Users need to have **an implementation for each** PDE
- **Implementation is complex**, out of the expertise of most domain scientists
- The users are more like "mathematicians" that want to **write PDEs with math notations**
- Further optimization required! (but the learning curve for C++ is steep…)

Implementation is complex. **~140 lines of C++ code** for the first tutorial in deal.ii*

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega$$

```
import GalerkinToolkit as GT
import ForwardDiff

domain = (0, 1, 0, 1, 0, 1)
cells = (10, 10, 10)
mesh = GT.cartesian_mesh(domain, cells)
Ω = GT.interior(mesh)
Γd = GT.boundary(mesh)
∇ = ForwardDiff.gradient
V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
dΩ = GT.measure(Ω, 2)
a = (u, v) → GT.∫(x → ∇(u, x) ⋅ ∇(v, x), dΩ)
l = v → GT.∫(x → v(x), dΩ)
p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
```

In GalerkinToolkit, we want ~10 lines of code for the complete problem

* https://dealii.org/current/doxygen/deal.II/step_3.html

# Existing Code generation strategy

- Domain Specific Languages (**DSLs**): Close to the mathematical notation of PDE
- Limitation: **Abstraction difficult to break**
- An example in UFL (with FEniCSx in Python):

```
f = fem.Function(V)
f.interpolate(lambda x: -D * k * (k-1) * ((x[0]+x[1]+x[2]) ** (k-2)))
du = ufl.TrialFunction(V)
dv = ufl.TestFunction(V)
a = ufl.dot(ufl.grad(du), ufl.grad(dv)) * ufl.dx
L = f * dv * ufl.dx
```

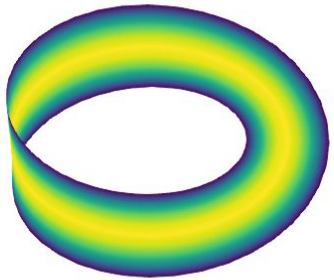
x is actually a numpy array

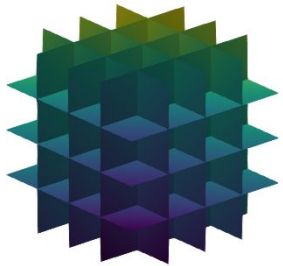
Everything in the UFL package ufl.*

Hard to include user-defined types or functions!

# Background: Why a new code generation library in Julia?

- We propose a new GalerkinToolkit to:
  - Solve the **2-language problem**
  - A **new code generation strategy** (see next slides)
  - Prove that **code generation is more efficient** than lazy array strategies in FEM, learning from the experience of Gridap
  - Be ambitious to make our library **general,** supporting a wide range of multilinear forms
  - Code generation beyond the assembly of multi-linear forms
  - Further development: be **portable** (CPUs and GPUs) and **cross-platform**



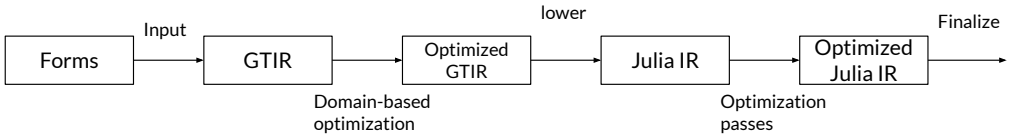


# Overview of code generation in GalerkinToolkit

```
1  import GalerkinToolkit as GT
2  import ForwardDiff
3
4  mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5  Ω = GT.interior(mesh)
6  Γd = GT.boundary(mesh)
7  ∇ = ForwardDiff.gradient
8  V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
9  dΩ = GT.measure(Ω, 2)
10 a = (u, v) -> GT.∫(x -> ∇(u, x) ⋅ ∇(v, x), dΩ)
11 l = v -> GT.∫(x -> v(x), dΩ)
12 p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
13
```

- Partially symbolic representation
- Optimizations
  - Domain-specific
  - General purpose passes based on assumptions & constraints
- Practical benefits with the Julia programming language
  - Efficient **Just-in-time compiler** by Julia
  - Close to math expression, many math operations can be reused
  - Built-in **meta-programming feature**
  - **Type inferencing** in Julia simplifies our code generation

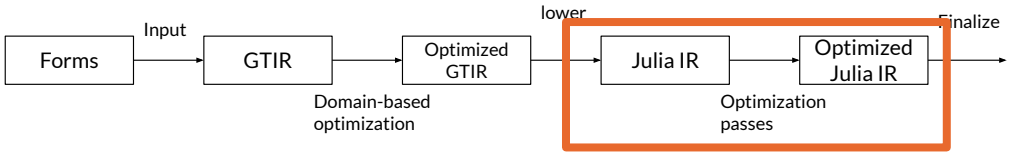
# Our code generation in GalerkinToolkit




```
1   import GalerkinToolkit as GT
2   import ForwardDiff
3
4   mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5   Ω = GT.interior(mesh)
6   Γd = GT.boundary(mesh)
7   ∇ = ForwardDiff.gradient
8   V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
9   dΩ = GT.measure(Ω, 2)
10  a = (u, v) -> GT.∫(x -> ∇(u, x) ⋅ ∇(v, x), dΩ)
11  l = v -> GT.∫(x -> v(x), dΩ)
12  p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
13
```

```
proto_41 = Val(true)
proto_42 = shape_function_accessor(ForwardDiff.gradient, captured_arg_2, captured_arg_4, proto_41)
proto_43 = proto_42(1, 1)
proto_44 = jacobian_accessor(captured_arg_4, Val{3}())
proto_45 = proto_44(1, 1)
proto_46 = proto_45(1)
proto_47 = proto_43(1, proto_46)
proto_48 = proto_47(1)
var_60 = typeof(proto_48)
var_61 = GT.field(captured_arg_2, 1)
var_62 = GT.max_num_reference_dofs(var_61)
var_63 = max(var_62)
var_13 = zeros(var_60, var_63)
domain = GT.domain(captured_arg_4)
T = eltype(alloc)
z = zero(T)
nfaces = GT.num_faces(domain)
face_npoints = GT.num_points_accessor(captured_arg_4)
var_str_init_1_1_1_1 = "be_for_$(1)_$(1)_$(1)_$(1)"
be_1_1_1_1_1 = alloc_zeros(var_str_init_1_1_1_1, T, var_62, var_62)
var_5 = GT.dofs_accessor(var_61, domain)
var_17 = weight_accessor(captured_arg_4)
for domain_face = 1:nfaces
    npoints = face_npoints(domain_face)
    fill!(be_1_1_1_1_1, z)
    dofs_trial_1_1_1 = var_5(domain_face, 1)
    ndofs_trial_1_1_1 = length(dofs_trial_1_1_1)
    var_8 = proto_42(domain_face, 1)
    var_10 = proto_44(domain_face, 1)
    var_18 = var_17(domain_face)
    for point = 1:npoints
        var_11 = var_10(point)
        var_12 = var_8(point, var_11)
        var_19 = var_18(point, var_11)
        for dof_2 = 1:ndofs_trial_1_1_1
            var_13[dof_2] = var_12(dof_2)
        end
        for dof_1 = 1:ndofs_trial_1_1_1
            var_14 = var_13[dof_1]
            for dof_2 = 1:ndofs_trial_1_1_1
                var_64 = var_13[dof_2]
                var_15 = (LinearAlgebra.dot)(var_64, var_14)
                var_16 = (*)(captured_arg_1, var_15)
                v_1_1_1_1 = (*)(var_16, var_19)
                be_1_1_1_1_1[dof_2, dof_1] += v_1_1_1_1
            end
        end
    end
    GT.contribute!(alloc, be_1_1_1_1_1, dofs_trial_1_1_1, dofs_trial_1_1_1, 1, 1)
end
```

# Our code generation in GalerkinToolkit




```
1   import GalerkinToolkit as GT
2   import ForwardDiff
3
4   mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5   Ω = GT.interior(mesh)
6   Γd = GT.boundary(mesh)
7   ∇ = ForwardDiff.gradient
8   V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
9   dΩ = GT.measure(Ω, 2)
10  a = (u, v) -> GT.∫(x -> ∇(u, x) ⋅ ∇(v, x), dΩ)
11  l = v -> GT.∫(x -> v(x), dΩ)
12  p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
13
```

```
proto_41 = Val(true)
proto_42 = shape_function_accessor(ForwardDiff.gradient, captured_arg_2, captured_arg_4, proto_41)
proto_43 = proto_42(1, 1)
proto_44 = jacobian_accessor(captured_arg_4, Val{3}())
proto_45 = proto_44(1, 1)
proto_46 = proto_45(1)
proto_47 = proto_43(1, proto_46)
proto_48 = proto_47(1)
var_60 = typeof(proto_48)
var_61 = GT.field(captured_arg_2, 1)
var_62 = GT.max_num_reference_dofs(var_61)
var_63 = max(var_62)
var_13 = zeros(var_60, var_63)
domain = GT.domain(captured_arg_4)
T = eltype(alloc)
z = zero(T)
nfaces = GT.num_faces(domain)
face_npoints = GT.num_points_accessor(captured_arg_4)
var_str_init_1_1_1_1 = "be_for_$(1)_$(1)_$(1)_$(1)"
be_1_1_1_1_1 = alloc_zeros(var_str_init_1_1_1_1, T, var_62, var_62)
var_5 = GT.dofs_accessor(var_61, domain)
var_17 = weight_accessor(captured_arg_4)
for domain_face = 1:nfaces
    npoints = face_npoints(domain_face)
    fill!(be_1_1_1_1_1, z)
    dofs_trial_1_1_1 = var_5(domain_face, 1)
    ndofs_trial_1_1_1 = length(dofs_trial_1_1_1)
    var_8 = proto_42(domain_face, 1)
    var_10 = proto_44(domain_face, 1)
    var_18 = var_17(domain_face)
    for point = 1:npoints
        var_11 = var_10(point)
        var_12 = var_8(point, var_11)
        var_19 = var_18(point, var_11)
        for dof_2 = 1:ndofs_trial_1_1_1
            var_13[dof_2] = var_12(dof_2)
        end
        for dof_1 = 1:ndofs_trial_1_1_1
            var_14 = var_13[dof_1]
            for dof_2 = 1:ndofs_trial_1_1_1
                var_64 = var_13[dof_2]
                var_15 = (LinearAlgebra.dot)(var_64, var_14)
                var_16 = (*)(captured_arg_1, var_15)
                v_1_1_1_1 = (*)(var_16, var_19)
                be_1_1_1_1_1[dof_2, dof_1] += v_1_1_1_1
            end
        end
    end
    GT.contribute!(alloc, be_1_1_1_1_1, dofs_trial_1_1_1, dofs_trial_1_1_1, 1, 1)
end
```

# METHODOLOGY: TEMPLATE

- The template: inject code in the **innermost loop** (complete but far from optimized!)

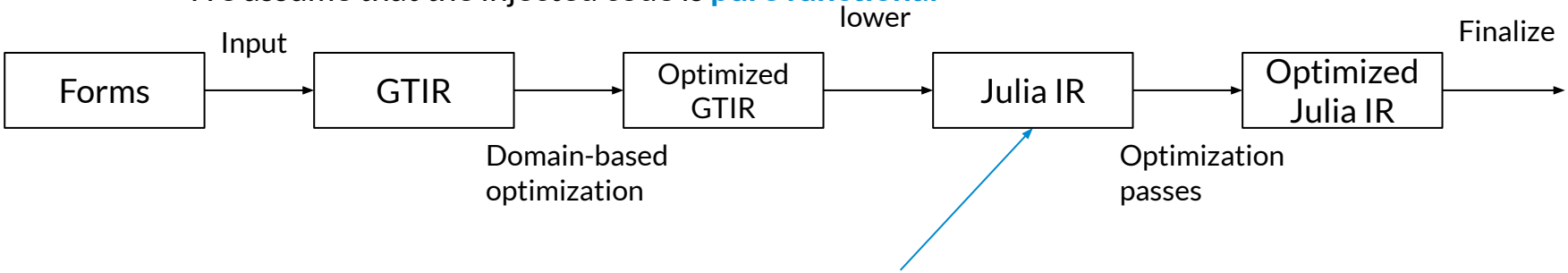
```
#u_i, v_j are shape functions Ω → [0, 1]
for f ∈ (1, . . . nf ) do
    A = 0 # a zero local matrix
    for p ∈ (1, . . . np) do
        x = coordinate(f, p)
        for i ∈ (1, . . . ni) do
            for j ∈ (1, . . . nj) do
                A[i, j] += ∇u_i(x) · ∇v_j(x)
            end for
        end for
    end for
end for
```

```
1   import GalerkinToolkit as GT
2   import ForwardDiff
3
4   mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5   Ω = GT.interior(mesh)
6   Γd = GT.boundary(mesh)
7   ∇ = ForwardDiff.gradient
8   V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
9   dΩ = GT.measure(Ω, 2)
10  a = (u, v) -> GT.∫(x -> ∇(u, x) ⋅ ∇(v, x)) dΩ)
11  l = v -> GT.∫(x -> v(x), dΩ)
12  p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
13
```

In the most general case, 8 nested loops , and the code injected can be complex

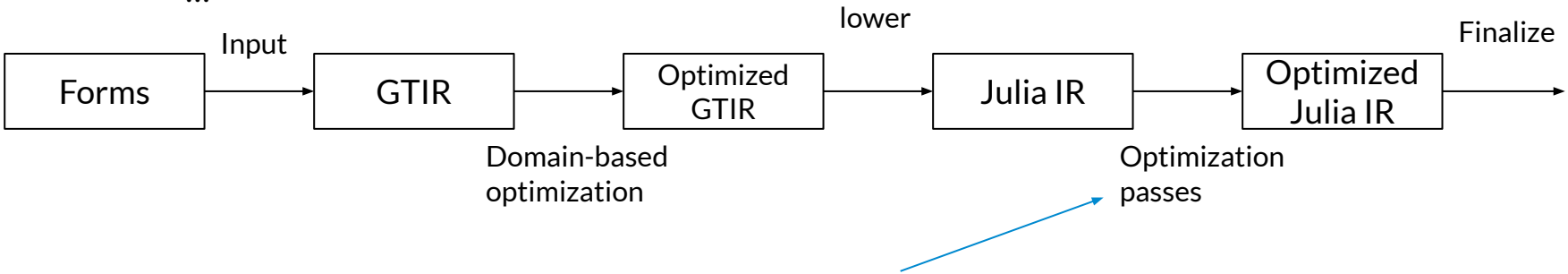
# METHODOLOGY: TRANSPILATION PASSES

- Julia IR: a subset of Julia
- Why Julia IR?
  - We need to have **additional assumptions** (not supported by a general compiler)
  - The template: inject code in the **innermost loop**, and then optimize
  - The template only uses a subset of Julia syntax:
    - Array indexed by loop variables or constants
    - Perfectly nested loops, so no continue/break/goto…
    - No variable re-definition, but accumulations are allowed
    - …
  - We assume that the injected code is **pure functional**



# METHODOLOGY: TRANSPILATION PASSES

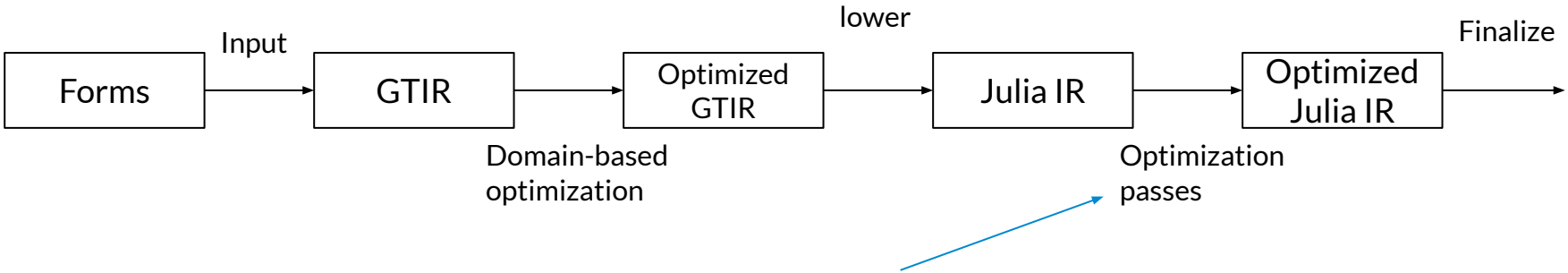
- Key passes
  - **Loop-invariant code motion (binomial-tree structured)**
  - Loop unrolling
  - Constant folding
  - Common subexpression elimination
- Pre/post processing
  - Flatten
  - Scalar replacement: a[1], a[2] -> a_1, a_2
  - Dead code elimination
  - …



# METHODOLOGY: LOOP-INVARIANT CODE MOTION

```
for i ∈ (1, . . . ni ) do
    for j ∈ (1, . . . nj) do
        for k ∈ (1, . . . nk) do
            v += f1(j) * f2(i, j) + f3(k) * f4(i, k)
        end for
    end for
end for
```

What if there is something that depends on *i* and *k* but not *j*?



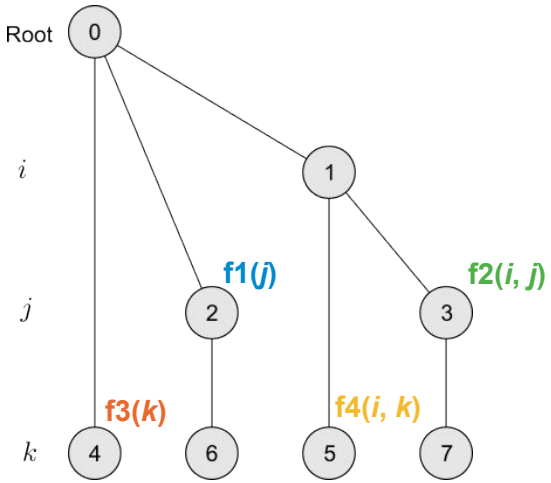
# METHODOLOGY: LOOP-INVARIANT CODE MOTION (OBJECTIVE)

```
for i ∈ (1, . . . ni ) do
        for j ∈ (1, . . . nj) do
                for k ∈ (1, . . . nk) do
                        v += f1(j) * f2(i, j) +
                                f3(k) * f4(i, k)
                end for
        end for
end for
```

```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
for j ∈ (1, . . . nj) do
        v2[j] = f1(j)
end for
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
        for j ∈ (1, . . . nj) do
                v6 = v2[j] * f2(i, j)
                for k ∈ (1, . . . nk) do
                        v += v6 * v5[k]
                end for
        end for
end for
```

# METHODOLOGY: LOOP-INVARIANT CODE MOTION

Binomial tree representing the dependency of expressions

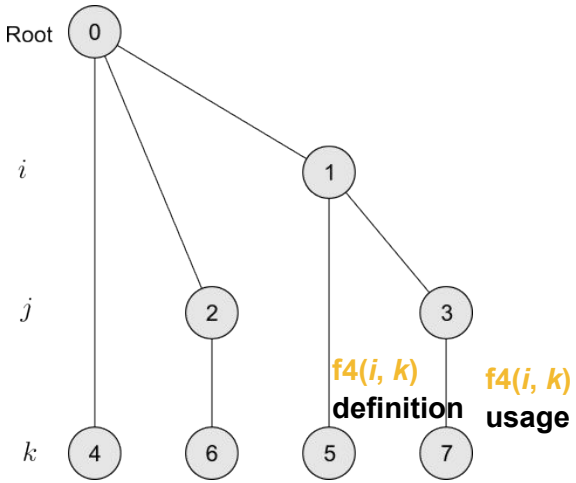



Bitmap representation of node ID:
i: $(001)_2$ j: $(010)_2$ k: $(100)_2$

```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
for j ∈ (1, . . . nj) do
        v2[j] = f1(j)
end for
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
         for j ∈ (1, . . . nj) do
                  v6 = v2[j] * f2(i, j)
                  for k ∈ (1, . . . nk) do
                           v += v6 * v5[k]
                  end for
         end for
end for
```

# METHODOLOGY: LOOP-INVARIANT CODE MOTION

How do we know the shape of temporary arrays?
With the Lowest common ancestor (LCA) of definition of usage (path from LCA to Definiton)




Definition: $(101)_2$ Usage: $(111)_2$

LCA: $(001)_2$

Shape: $(101)_2$ - $(001)_2$ = $(100)_2$ , represents (i, )

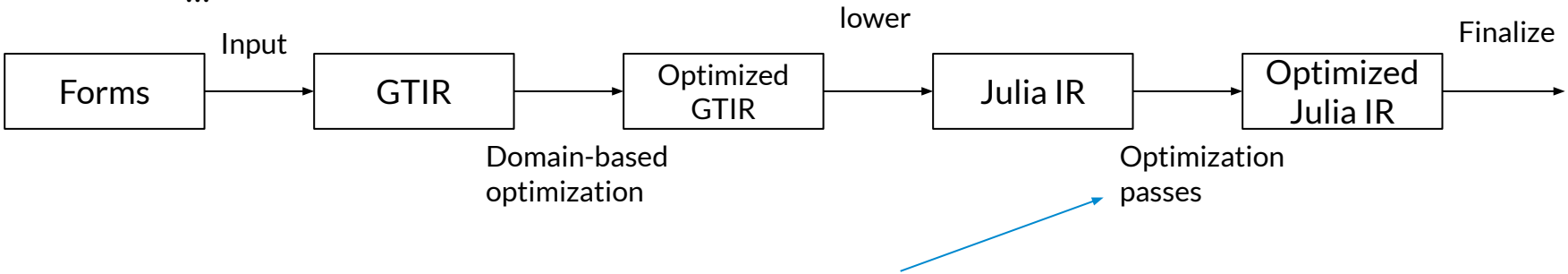
```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
      v1[k] = f3(k)
end for
for j ∈ (1, . . . nj) do
      v2[j] = f1(j)
end for
for i ∈ (1, . . . ni ) do
      for k ∈ (1, . . . nk) do
            v5[k] = v1[k] * f4(i, k)
      end for
       for j ∈ (1, . . . nj) do
             v6 = v2[j] * f2(i, j)
            for k ∈ (1, . . . nk) do
                  v += v6 * v5[k]
            end for
       end for
end for
```

# METHODOLOGY: TRANSPILATION PASSES

- Key passes
  - Loop-invariant code motion
  - **Loop unrolling**
  - **Constant folding**
  - Common subexpression elimination

⇨ For multi-field or mixed-dimension problems

- Pre/post processing
  - Flatten
  - Scalar replacement: a[1], a[2] -> a_1, a_2
  - Dead code elimination
  - …



# METHODOLOGY: LOOP UNROLLING

```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
for j ∈ (1,2) do            # here we let nj = 2
        v2[j] = f1(j)
end for
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
         for j ∈ (1, 2) do
                  v6 = v2[j] * f2(i, j)
                 for k ∈ (1, . . . nk) do
                          v += v6 * v5[k]
                 end for
         end for
end for
```

```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
v2[1] = f1(1)
v2[2] = f1(2)
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
        v6_1 = v2[1] * f2(i, 1)
        for k ∈ (1, . . . nk) do
                v += v6_1 * v5[k]
         end for
        v6_2 = v2[2] * f2(i, 2)
        for k ∈ (1, . . . nk) do
                v += v6_2 * v5[k]
         end for
end for
```

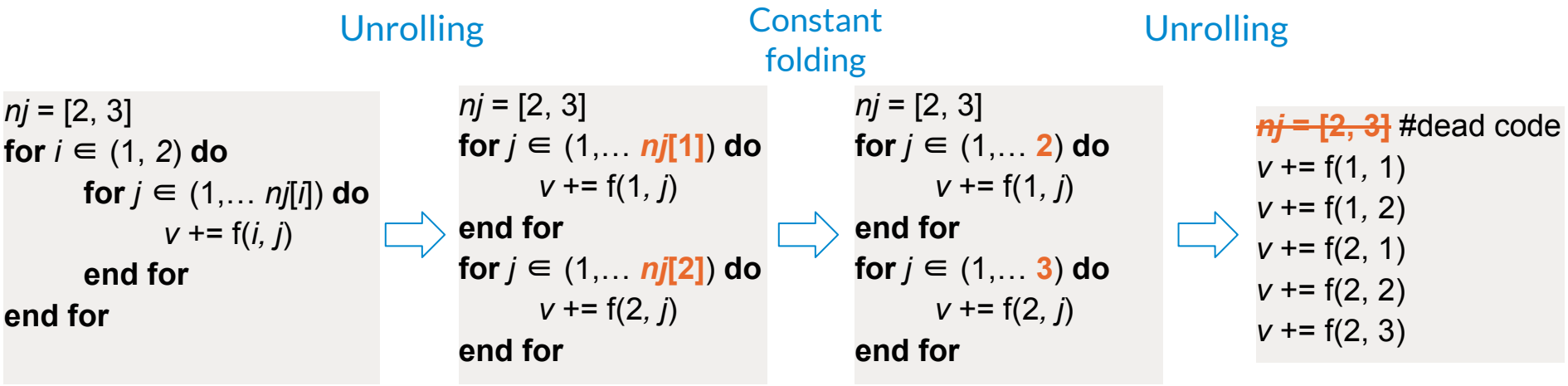
# METHODOLOGY: LOOP UNROLLING & SCALAR REPLACEMENT

```
… Allocate v1[nk], v2[nj], v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
for j ∈ (1,2) do            # here we let nj = 2
        v2[j] = f1(j)
end for
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
         for j ∈ (1, 2) do
                  v6 = v2[j] * f2(i, j)
                 for k ∈ (1, . . . nk) do
                          v += v6 * v5[k]
                 end for
         end for
end for
```

with scalar replacement

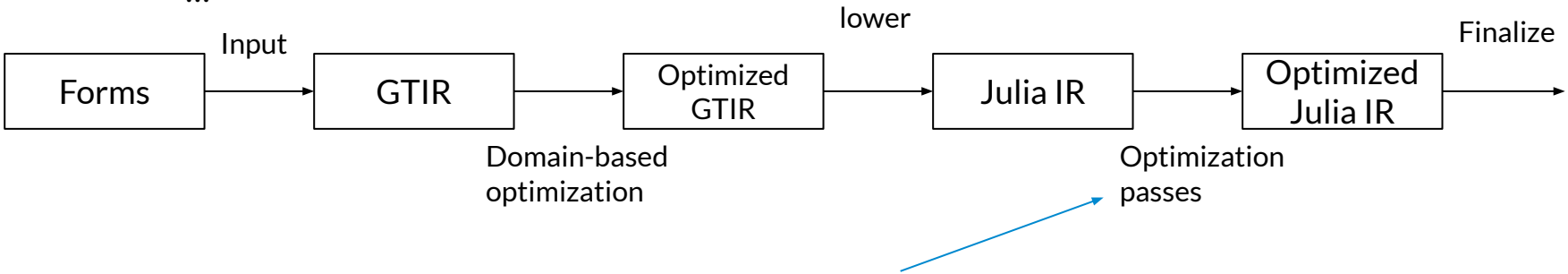
```
… Allocate v1[nk], ~~v2[nj],~~ v5[nk]
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
v2_1 = f1(1)
v2_2 = f1(2)
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
        v6_1 = v2_1 * f2(i, 1)
        for k ∈ (1, . . . nk) do
                 v += v6_1 * v5[k]
         end for
         v6_2 = v2_2 * f2(i, 2)
        for k ∈ (1, . . . nk) do
                 v += v6_2 * v5[k]
         end for
end for
```

# METHODOLOGY: CONSTANT FOLDING AFTER UNROLLING

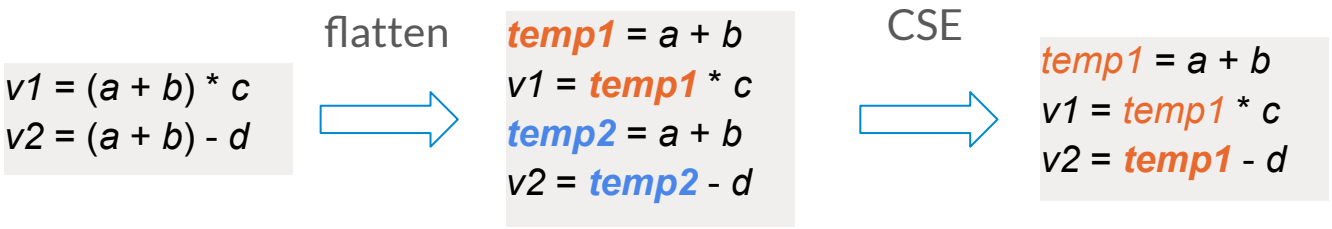


# METHODOLOGY: TRANSPILATION PASSES

- Key passes
  - Loop-invariant code motion
  - Loop unrolling
  - Constant folding
  - **Common subexpression elimination**
- Pre/post processing
  - Flatten
  - Scalar replacement: a[1], a[2] -> a_1, a_2
  - Dead code elimination
  - …

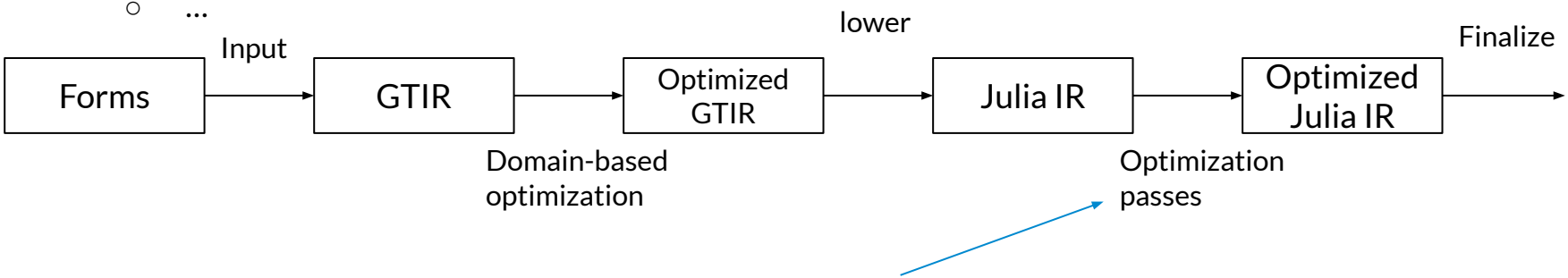


# METHODOLOGY: COMMON SUBEXPRESSION ELIMINATION (WITH FLATTEN)



# METHODOLOGY: TRANSPILATION PASSES

- Key passes
  - **Generalized** Loop-invariant code motion (binomial-tree structured) that is **not limited to perfectly nested loops**
  - Loop unrolling
  - Constant folding
  - Common subexpression elimination
- Pre/post processing
  - Flatten
  - Scalar replacement: a[1], a[2] -> a_1, a_2
  - Dead code elimination
  - …

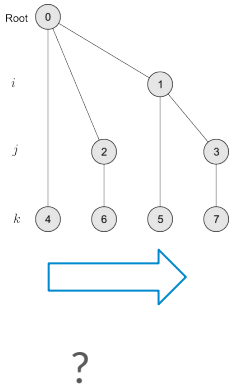


# Loop-invariant code motion (also works after unrolling)

```
for i ∈ (1, . . . ni ) do
        for j ∈ (1, 2) do
                for k ∈ (1, . . . nk) do
                        v += f1(j) * f2(i, j) +
                                f3(k) * f4(i, k)
                end for
        end for
end for
```

This **does not follow the binomial tree structure** any more

```
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v += f1(1) * f2(i, 1) +
                        f3(k) * f4(i, k)
        end for
        for k ∈ (1, . . . nk) do
                v += f1(2) * f2(i, 2) +
                        f3(k) * f4(i, k)
        end for
end for
```

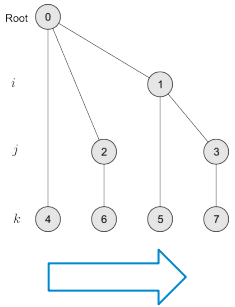



How can we perform loop-invariant code motion with the unrolled loops?

# LOOP-INVARIANT CODE MOTION (ALSO WORKS AFTER UNROLLING)

```
for i ∈ (1, . . . ni ) do
        for j ∈ (1, 2) do
                for k ∈ (1, . . . nk) do
                        v += f1(j) * f2(i, j) +
                                f3(k) * f4(i, k)
                end for
        end for
end for
```

```
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v += f1(1) * f2(i, 1) +
                        f3(k) * f4(i, k)
        end for
        for k ∈ (1, . . . nk) do
                v += f1(2) * f2(i, 2) +
                        f3(k) * f4(i, k)
        end for
end for
```



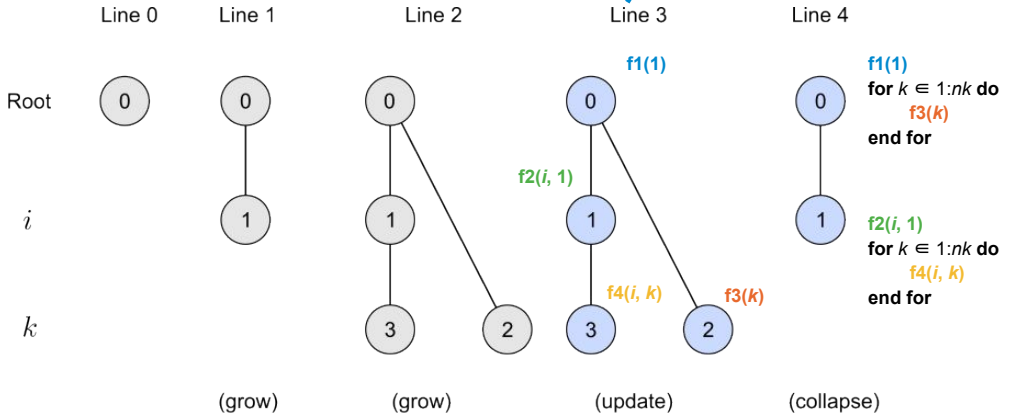

Build the binomial tree dynamically!

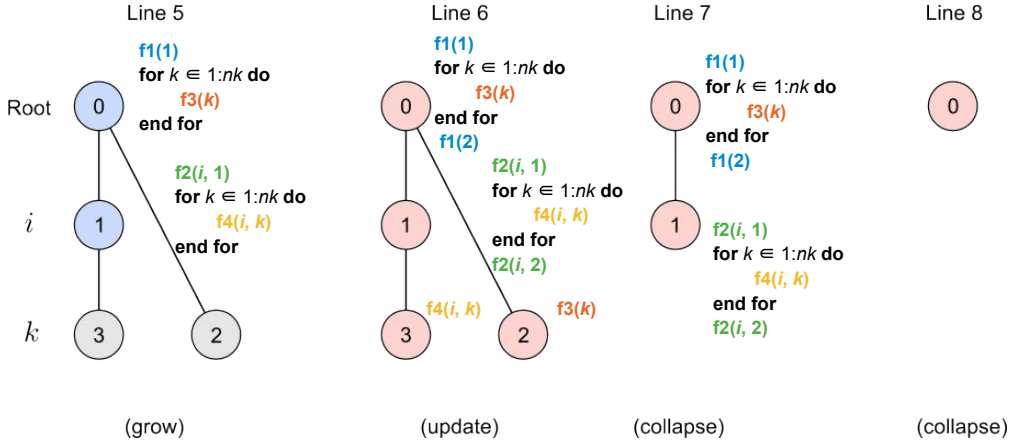
# LOOP-INVARIANT CODE MOTION (ALSO WORKS AFTER UNROLLING)

Dynamic binomial tree with **grow** and **collapse** operations

(skipping temporary variables and accumulation)

```
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v += f1(1) * f2(i, 1) +
                        f3(k) * f4(i, k)
        end for
        for k ∈ (1, . . . nk) do
                v += f1(2) * f2(i, 2) +
                        f3(k) * f4(i, k)
        end for
end for
```



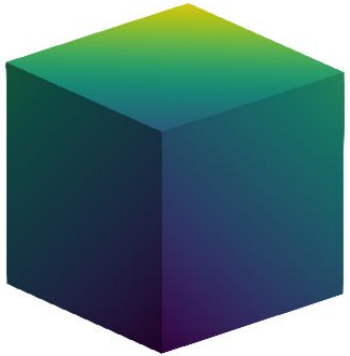
# LOOP-INVARIANT CODE MOTION (ALSO WORKS AFTER UNROLLING)

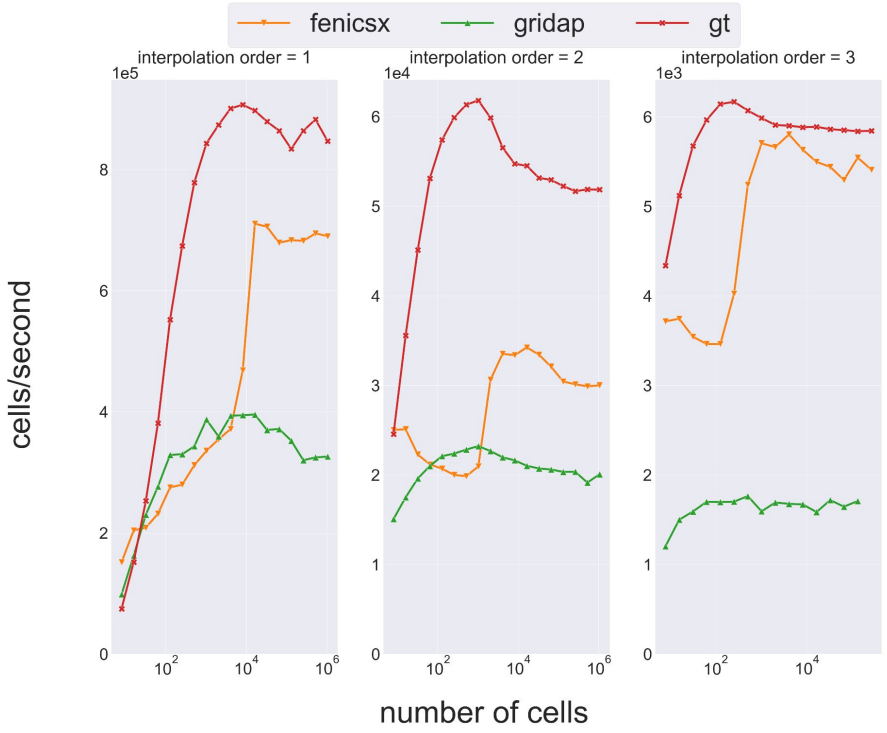
```
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v += f1(1) * f2(i, 1) +
                        f3(k) * f4(i, k)
        end for
        for k ∈ (1, . . . nk) do
                v += f1(2) * f2(i, 2) +
                        f3(k) * f4(i, k)
        end for
end for
```

```
… Allocate v1[nk], v5[nk]
v2_1 = f1(1)
for k ∈ (1, . . . nk) do
        v1[k] = f3(k)
end for
v2_2 = f1(2)
for i ∈ (1, . . . ni ) do
        for k ∈ (1, . . . nk) do
                v5[k] = v1[k] * f4(i, k)
        end for
        v6_1 = v2_1 * f2(i, 1)
        for k ∈ (1, . . . nk) do
                v += v6_1 * v5[k]
        end for
        v6_2 = v2_2 * f2(i, 2)
        for k ∈ (1, . . . nk) do
                v += v6_2 * v5[k]
        end for
end for
```

# BENCHMARKING: POISSON EQUATION

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega = \int_\Omega f v \, \mathrm{d}\Omega$$





# BENCHMARKING: POISSON EQUATION WITH DISCONTINUOUS GALERKIN

- The problem:

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega + \int_{\partial\Omega} \left( -(\nabla u \cdot n)\, v - (\nabla v \cdot n)\, u + \frac{\gamma}{h} v u \right) \mathrm{d}\Gamma +$$

$$\int_\Omega \left( -\mathrm{jump}\,(v, n) \cdot \mathrm{mean}\,(\nabla u) - \mathrm{jump}\,(u, n) \cdot \mathrm{mean}\,(\nabla v) + \frac{\gamma}{h} \mathrm{jump}\,(u, n) \cdot \mathrm{jump}\,(v, n) \right) \mathrm{d}\Lambda =$$

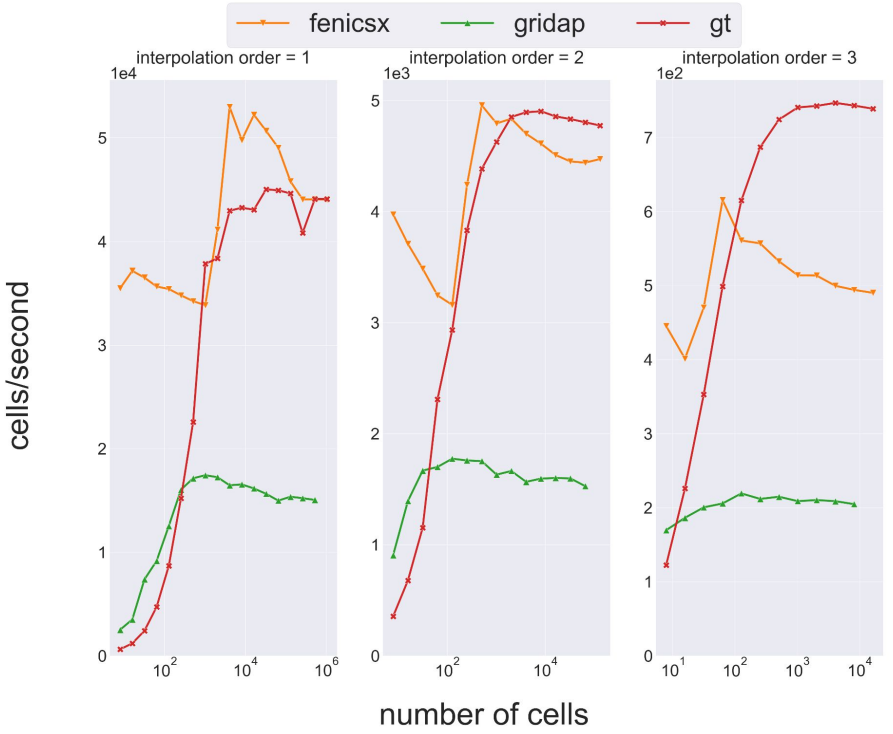
$$\int_\Omega f v \, \mathrm{d}\Omega + \int_{\partial\Omega} \left( \frac{\gamma}{h} g v - n \cdot \nabla v g \right) \mathrm{d}\Gamma$$

# BENCHMARKING: POISSON EQUATION WITH DISCONTINUOUS GALERKIN

$$\int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\Omega + \int_{\partial\Omega} \left( -(\nabla u \cdot n) v - (\nabla v \cdot n) u + \frac{\gamma}{h} v u \right) \mathrm{d}\Gamma +$$

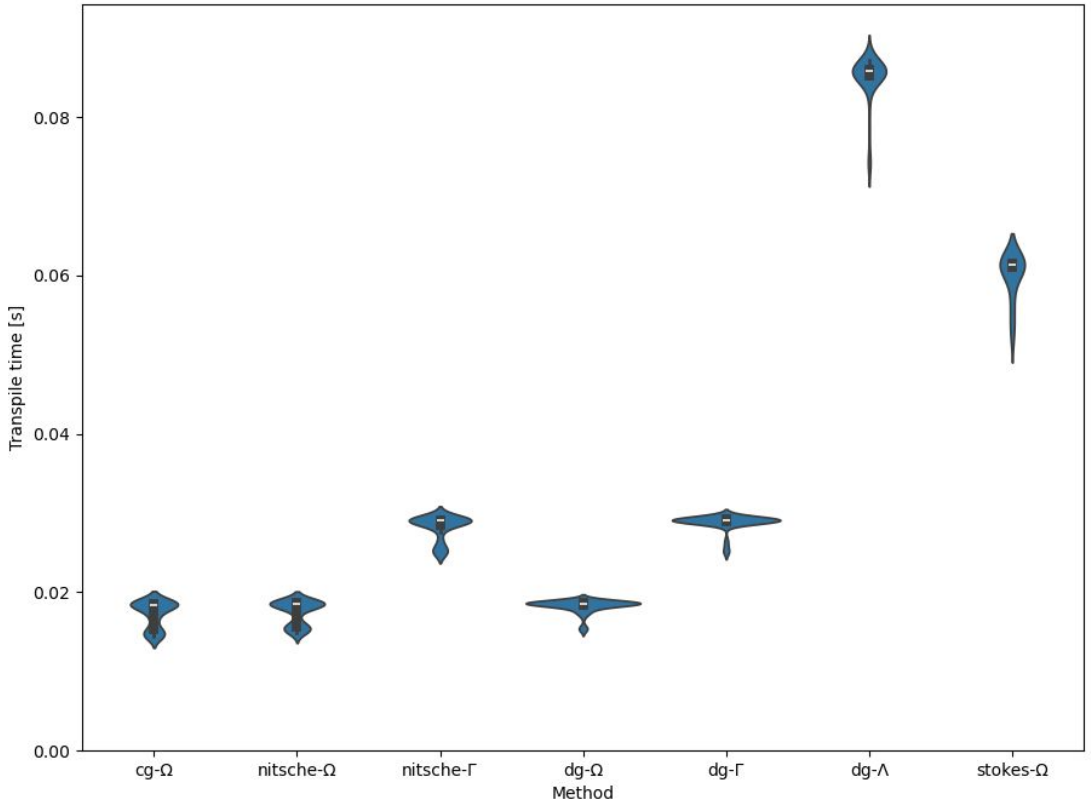
$$\int_\Omega \left( -\mathrm{jump}\,(v, n) \cdot \mathrm{mean}\,(\nabla u) - \mathrm{jump}\,(u, n) \cdot \mathrm{mean}\,(\nabla v) + \frac{\gamma}{h} \mathrm{jump}\,(u \right.$$

$$\int_\Omega f v \, \mathrm{d}\Omega + \int_{\partial\Omega} \left( \frac{\gamma}{h} g v - n \cdot \nabla v g \right) \mathrm{d}\Gamma$$



# BENCHMARKING

Code generation time: milliseconds



# Summary

- New code generation methodology
  - 1-language solution with Julia without a DSL
  - Optimization passes based on the global loop
  - Loop-invariant code motion strategy designed for the problem
- Performance
  - (re-)assembly competitive with respective to existing libraries
  - Code generation (transpile) in milliseconds, and independent of the mesh size

# Future works

- Working in progress
  - GPU assembly low level glue code
  - Sequential code optimizations
    - Tiling
    - AVX512 support
    - Element or problem specific
      - Structured mesh
      - Simplex element optimization
      - Sum factorization for high-order elements
    - Equation specific
      - Symmetric optimization
- Future plans
  - GPU code generation (that is why we want to optimize global loops)
  - Automatic differentiation
  - Distributed on clusters

# Questions?