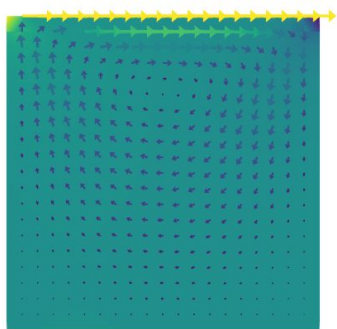
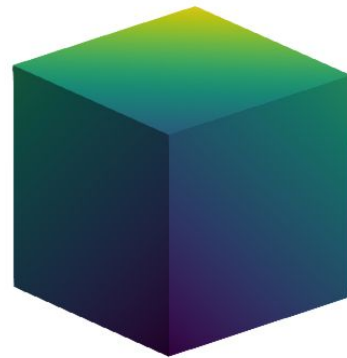


# EXPRESSIVE AND EFFICIENT FINITE ELEMENT CODE GENERATION IN JULIA: THE PROGRESS OF GALERKINTOOLKIT FORM COMPILER



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...)

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

```
1 #include <deal.II/grid/1
2 #include <deal.II/grid/54
3 #include <deal.II/fe/55
4 #include <deal.II/fe/56
5 #include <deal.II/fe/57
6 #include <deal.II/fe/58
7 #include <deal.II/fe/59
8 #include <deal.II/fe/60
9 #include <deal.II/fe/61
10 #include <deal.II/fe/62
11 #include <deal.II/fe/63
12 #include <deal.II/fe/64
13 #include <deal.II/fe/65
14 #include <deal.II/fe/66
15 #include <deal.II/fe/67
16 #include <deal.II/fe/68
17 #include <deal.II/fe/69
18 #include <deal.II/fe/70
19 #include <deal.II/fe/71
20 #include <deal.II/fe/72
21 #include <deal.II/fe/73
22 #include <deal.II/fe/74
23 #include <deal.II/fe/75
24 #include <deal.II/fe/76
25 #include <deal.II/fe/77
26 #include <deal.II/fe/78
27 #include <deal.II/fe/79
28 #include <deal.II/fe/80
29 #include <deal.II/fe/81
30 #include <deal.II/fe/82
31 #include <deal.II/fe/83
32 #include <deal.II/fe/84
33 #include <deal.II/fe/85
34 #include <deal.II/fe/86
35 #include <deal.II/fe/87
36 #include <deal.II/fe/88
37 #include <deal.II/fe/89
38 #include <deal.II/fe/90
39 #include <deal.II/fe/91
40 #include <deal.II/fe/92
41 #include <deal.II/fe/93
42 #include <deal.II/fe/94
43 #include <deal.II/fe/95
44 #include <deal.II/fe/96
45 #include <deal.II/fe/97
46 #include <deal.II/fe/98
47 #include <deal.II/fe/99
48 #include <deal.II/fe/100
49 #include <deal.II/fe/101
50 #include <deal.II/fe/102
51 #include <deal.II/fe/103
52 #include <deal.II/fe/104
53 #include <deal.II/fe/105
54 #include <deal.II/fe/106
55 #include <deal.II/fe/107
56 #include <deal.II/fe/108
57 #include <deal.II/fe/109
58 #include <deal.II/fe/110
59 #include <deal.II/fe/111
60 #include <deal.II/fe/112
61 #include <deal.II/fe/113
62 #include <deal.II/fe/114
63 #include <deal.II/fe/115
64 #include <deal.II/fe/116
65 #include <deal.II/fe/117
66 #include <deal.II/fe/118
67 #include <deal.II/fe/119
68 #include <deal.II/fe/120
69 #include <deal.II/fe/121
70 #include <deal.II/fe/122
71 #include <deal.II/fe/123
72 #include <deal.II/fe/124
73 #include <deal.II/fe/125
74 #include <deal.II/fe/126
75 #include <deal.II/fe/127
76 #include <deal.II/fe/128
77 #include <deal.II/fe/129
78 #include <deal.II/fe/130
79 #include <deal.II/fe/131
80 #include <deal.II/fe/132
81 #include <deal.II/fe/133
82 #include <deal.II/fe/134
83 #include <deal.II/fe/135
84 #include <deal.II/fe/136
85 #include <deal.II/fe/137
86 #include <deal.II/fe/138
87 #include <deal.II/fe/139
88 #include <deal.II/fe/140
89 #include <deal.II/fe/141
90 #include <deal.II/fe/142
91 #include <deal.II/fe/143
92 #include <deal.II/fe/144
93 #include <deal.II/fe/145
94 #include <deal.II/fe/146
95 #include <deal.II/fe/147
96 #include <deal.II/fe/148
97 #include <deal.II/fe/149
98 #include <deal.II/fe/150
99 #include <deal.II/fe/151
100 #include <deal.II/fe/152
101 #include <deal.II/fe/153
102 #include <deal.II/fe/154
103 #include <deal.II/fe/155
104 #include <deal.II/fe/156
105 #include <deal.II/fe/157
106 #include <deal.II/fe/158
107 #include <deal.II/fe/159
108 #include <deal.II/fe/160
109 #include <deal.II/fe/161
110 #include <deal.II/fe/162
111 #include <deal.II/fe/163
112 #include <deal.II/fe/164
113 #include <deal.II/fe/165
114 #include <deal.II/fe/166
115 #include <deal.II/fe/167
116 #include <deal.II/fe/168
117 #include <deal.II/fe/169
118 #include <deal.II/fe/170
119 #include <deal.II/fe/171
120 #include <deal.II/fe/172
121 #include <deal.II/fe/173
122 #include <deal.II/fe/174
123 #include <deal.II/fe/175
124 #include <deal.II/fe/176
125 #include <deal.II/fe/177
126 #include <deal.II/fe/178
127 #include <deal.II/fe/179
128 #include <deal.II/fe/180
129 #include <deal.II/fe/181
130 #include <deal.II/fe/182
131 #include <deal.II/fe/183
132 #include <deal.II/fe/184
133 #include <deal.II/fe/185
134 #include <deal.II/fe/186
135 #include <deal.II/fe/187
136 #include <deal.II/fe/188
137 #include <deal.II/fe/189
138 #include <deal.II/fe/190
139 #include <deal.II/fe/191
140 #include <deal.II/fe/192
141 #include <deal.II/fe/193
142 #include <deal.II/fe/194
143 #include <deal.II/fe/195
144 #include <deal.II/fe/196
145 #include <deal.II/fe/197
146 #include <deal.II/fe/198
147 #include <deal.II/fe/199
150 #include <deal.II/fe/200
```



```
import GalerkinToolkit as GT
import ForwardDiff

domain = (0, 1, 0, 1, 0, 1)
cells = (10, 10, 10)
mesh = GT.cartesian_mesh(domain, cells)
D = GT.interior(mesh)
fd = GT.boundary(mesh)
V = ForwardDiff.gradient
V = GT.lagrange_space(0, 1; dirichlet_boundary=fd)
dOmega = GT.measure(D, 2)
a = (u, v) -> GT.f(x -> V(u, x) * V(v, x), dOmega)
l = v -> GT.f(x -> v(x), dOmega)
p = GT.ScimlBase.LinearProblem(Float64, V, a, l)
```

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

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

\* [https://dealii.org/current/doxygen/deal.II/step\\_3.html](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
```

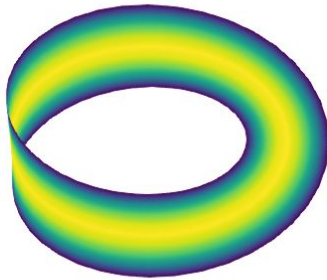
x is actually a  
numpy array

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**

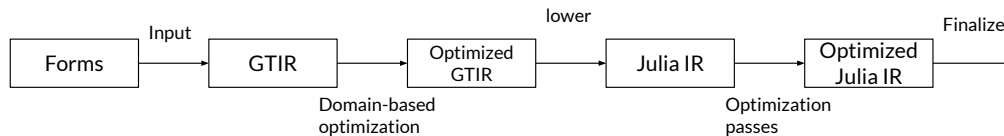


# OVERVIEW OF CODE GENERATION IN GALERKINTOOLKIT

- 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

```
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.f(x -> ∇(u, x) · ∇(v, x), dΩ)
11 l = v -> GT.f(x -> v(x), dΩ)
12 p = GT.SciMLBase_LinearProblem(Float64, V, a, l)
13
```

# OUR CODE GENERATION IN GALERKINTOOLKIT

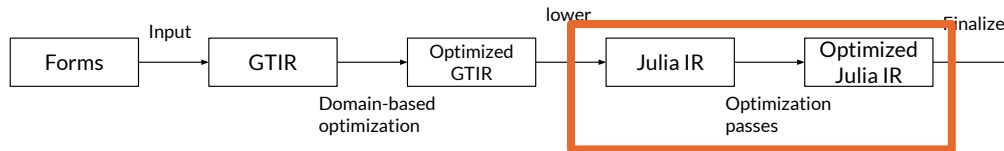


```
1 import GalerkinToolkit as GT
2 import ForwardDiff
3
4 mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5 Ω = GT.interior(mesh)
6 Γd = GT.boundary(mesh)
7 ∇ = ForwardDiff.gradient
8 V = GT.lagrange_space(Ω, 1; dirichlet_boundary=Γd)
9 dΩ = GT.measure(Ω, 2)
10 a = (u, v) -> GT.∫(x -> ∇(u, 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 = 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[dof_2, dof_1] += v_1_1_1_1
      end
    end
  end
end
end
GT.contribute!(alloc, be_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 = 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[dof_2, dof_1] += v_1_1_1_1
      end
    end
  end
end
end
GT.contribute!(alloc, be_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  $\Omega \rightarrow [0, 1]$ 
```

```
for f  $\in (1, \dots, nf)$  do
```

```
  A = 0 # a zero local matrix
```

```
  for p  $\in (1, \dots, np)$  do
```

```
    x = coordinate(f, p)
```

```
    for i  $\in (1, \dots, ni)$  do
```

```
      for j  $\in (1, \dots, nj)$  do
```

```
        A[i, j] +=  $\nabla u_i(x) \cdot \nabla v_j(x)$ 
```

```
      end for
```

```
    end for
```

```
  end for
```

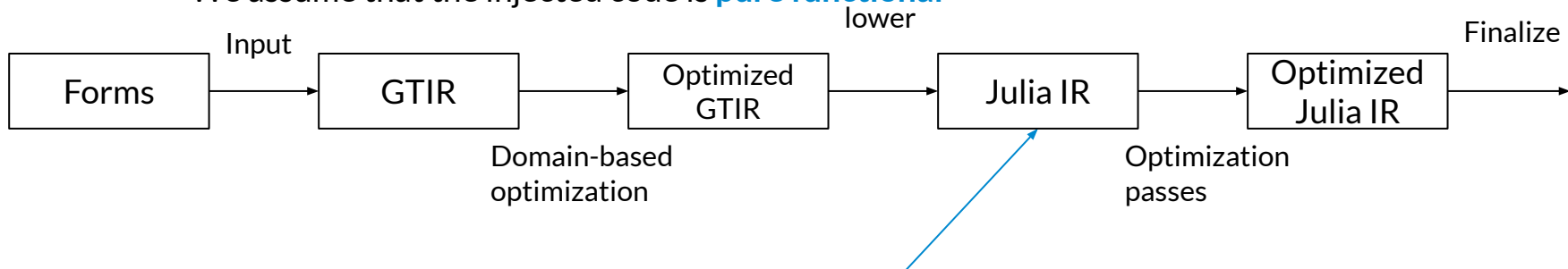
```
end for
```

```
1 import GalerkinToolkit as GT
2 import ForwardDiff
3
4 mesh = GT.cartesian_mesh((0, 1, 0, 1, 0, 1), (10, 10, 10))
5  $\Omega$  = GT.interior(mesh)
6  $\Gamma_d$  = GT.boundary(mesh)
7  $\nabla$  = ForwardDiff.gradient
8 V = GT.lagrange_space( $\Omega$ , 1; dirichlet_boundary= $\Gamma_d$ )
9 d $\Omega$  = GT.measure( $\Omega$ , 2)
10 a = (u, v) -> GT.j(x ->  $\nabla(u, x) \cdot \nabla(v, x)$ ) d $\Omega$ 
11 l = v -> GT.j(x -> v(x), d $\Omega$ )
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

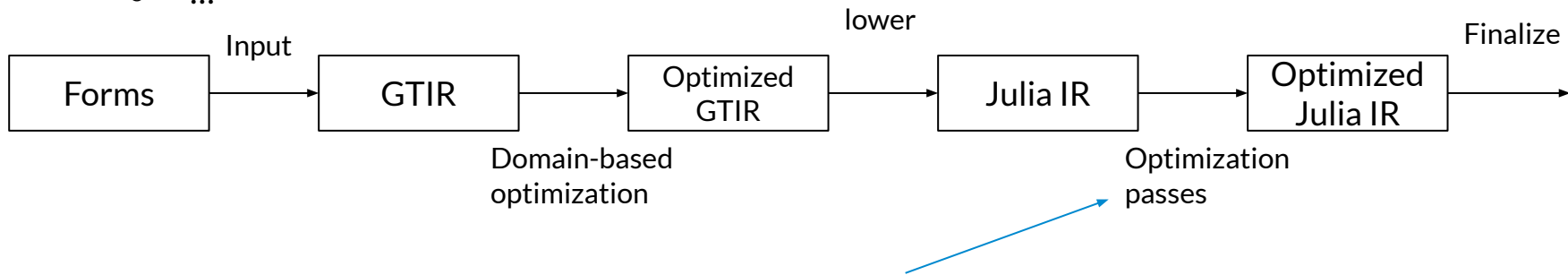
# METHODOLOGY: TRANSPILATION PASSES

- 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

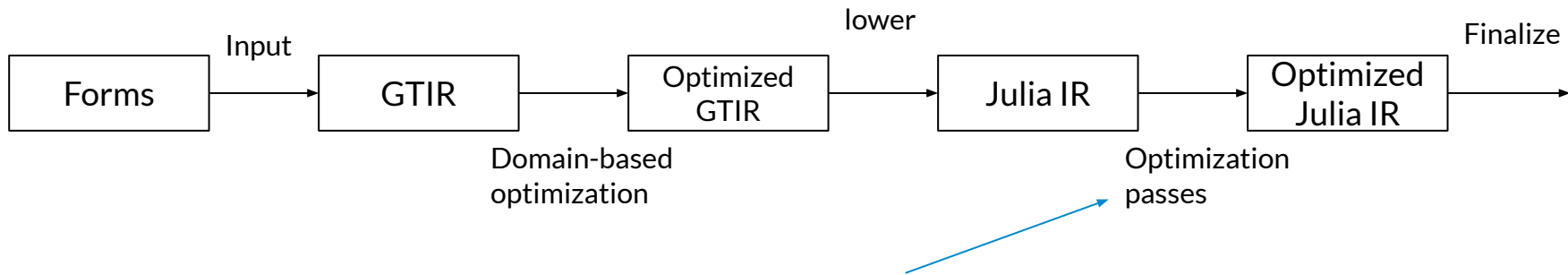
- Key passes
  - Loop-invariant code motion (binomial-tree structured)
  - Loop unrolling
  - Constant folding
  - Common subexpression elimination
- Pre/post processing
  - Flatten
  - Scalar replacement:  $a[1], a[2] \rightarrow a\_1, a\_2$
  - Dead code elimination
  - ...



# METHODOLOGY: LOOP-INVARIANT CODE MOTION

```
for  $i \in (1, \dots, ni)$  do
  for  $j \in (1, \dots, nj)$  do
    for  $k \in (1, \dots, nk)$  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)

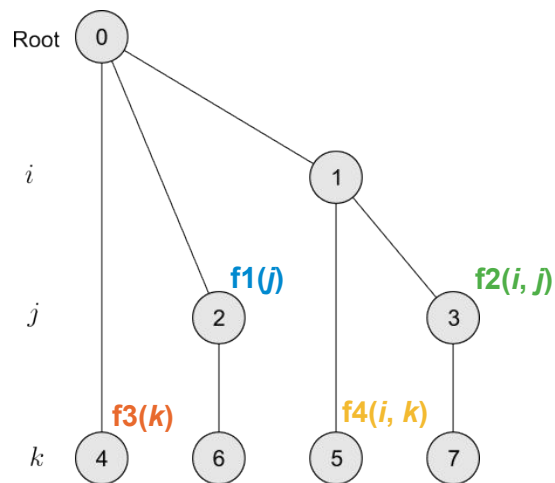
```
for i ∈ (1, ... ni) do
  for j ∈ (1, ... nj) do
    for k ∈ (1, ... nk) do
      v += f1(j) * f2(i, j) +
           f3(k) * f4(i, k)
    end for
  end for
end for
```



```
... 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

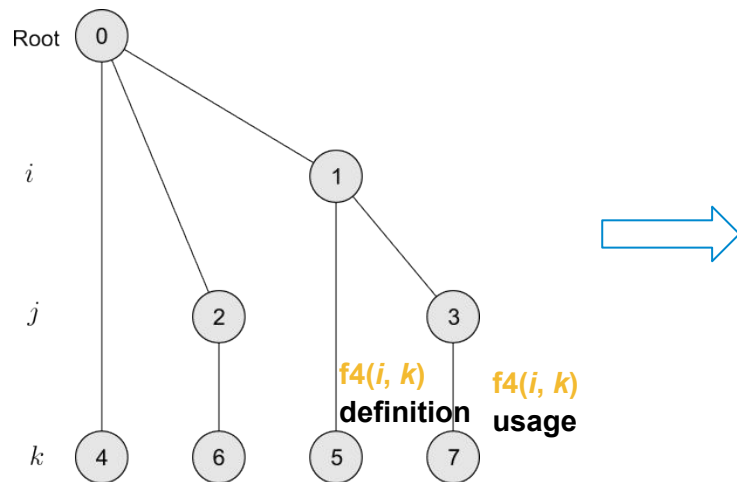


Bitmap representation of node ID:  
i:  $(001)_2$     j:  $(010)_2$     k:  $(100)_2$

```
... 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 Definition)



Definition:  $(101)_2$

Usage:  $(111)_2$

LCA:  $(001)_2$

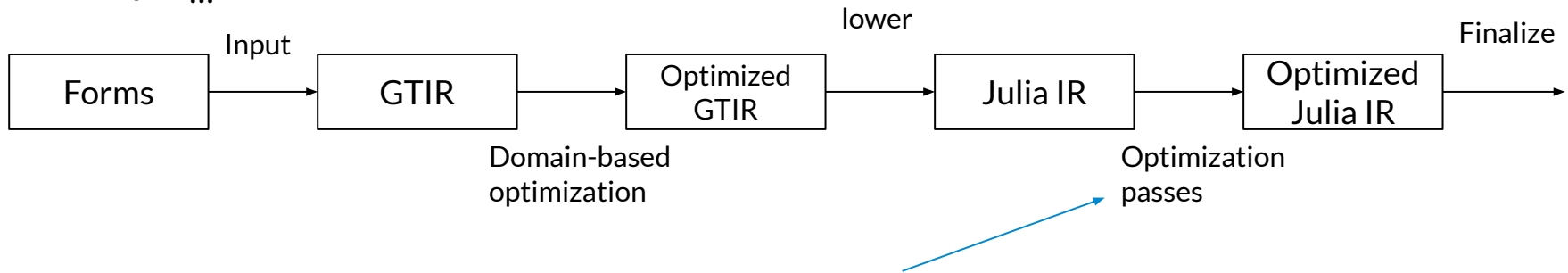
Shape:  $(101)_2 - (001)_2 = (100)_2$ , represents  $(i, )$

```

... 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**  $\Rightarrow$  For multi-field or mixed-dimension problems
  - Common subexpression elimination
- Pre/post processing
  - Flatten
  - Scalar replacement:  $a[1], a[2] \rightarrow a\_1, a\_2$
  - 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
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
end for
```

# METHODOLOGY: CONSTANT FOLDING AFTER UNROLLING

Unrolling

```
nj = [2, 3]
for i ∈ (1, 2) do
  for j ∈ (1, ... nj[i]) do
    v += f(i, j)
  end for
end for
```



```
nj = [2, 3]
for j ∈ (1, ... nj[1]) do
  v += f(1, j)
end for
for j ∈ (1, ... nj[2]) do
  v += f(2, j)
end for
```

Constant  
folding



```
nj = [2, 3]
for j ∈ (1, ... 2) do
  v += f(1, j)
end for
for j ∈ (1, ... 3) do
  v += f(2, j)
end for
```

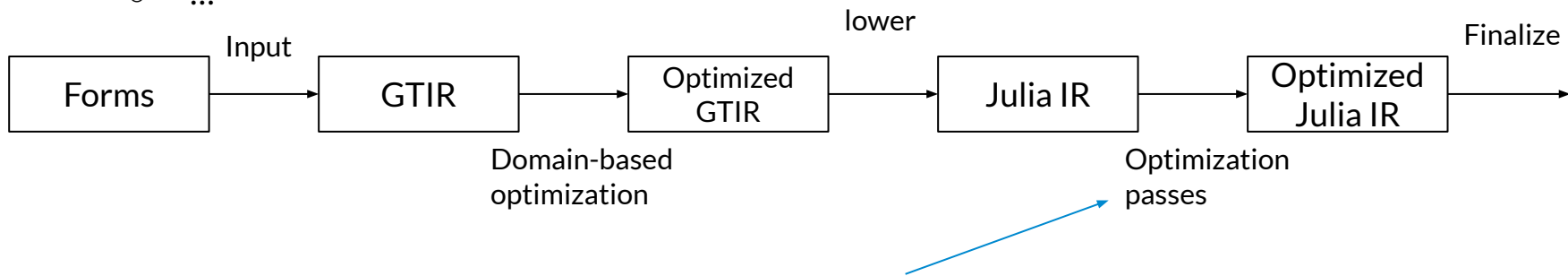
Unrolling



```
nj = [2, 3] #dead code
v += f(1, 1)
v += f(1, 2)
v += f(2, 1)
v += f(2, 2)
v += f(2, 3)
```

# METHODOLOGY: TRANSPILATION PASSES

- Key passes
  - Loop-invariant code motion
  - Loop unrolling
  - Constant folding
  - **Common subexpression elimination**
- Pre/post processing
  - Flatten
  - Scalar replacement:  $a[1], a[2] \rightarrow a\_1, a\_2$
  - Dead code elimination
  - ...



# METHODOLOGY: COMMON SUBEXPRESSION ELIMINATION (WITH FLATTEN)

```
v1 = (a + b) * c  
v2 = (a + b) - d
```

flatten



```
temp1 = a + b  
v1 = temp1 * c  
temp2 = a + b  
v2 = temp2 - d
```

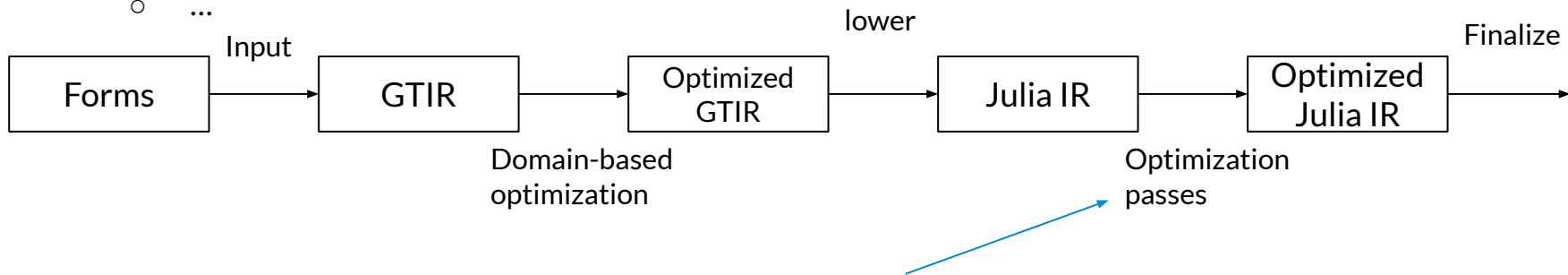
CSE



```
temp1 = a + b  
v1 = temp1 * c  
v2 = temp1 - d
```

# 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:  $a[1], a[2] \rightarrow a_1, a_2$
  - Dead code elimination
  - ...



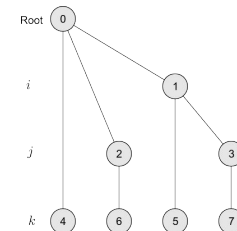
# LOOP-INVARIANT CODE MOTION (ALSO WORKS AFTER UNROLLING)

This does not follow the binomial tree structure any more

```
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
```



?

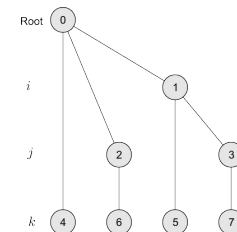
How can we perform loop-invariant code motion with the unrolled loops?

# 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
```



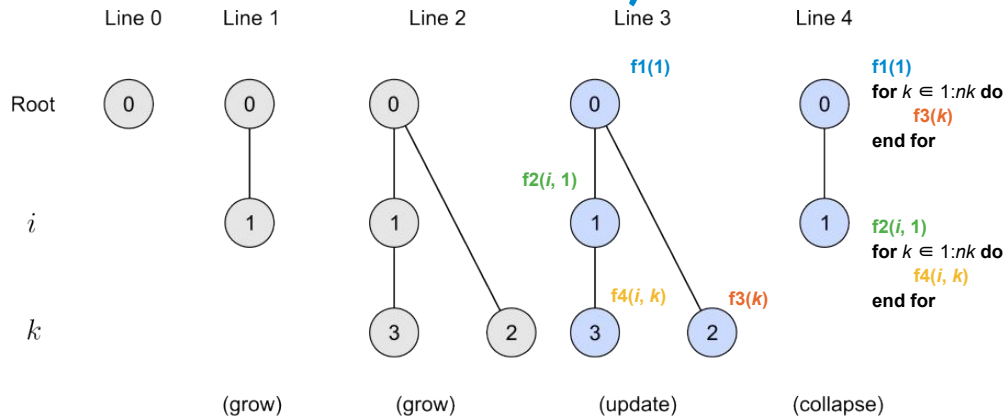
?

Build the binomial tree dynamically!

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

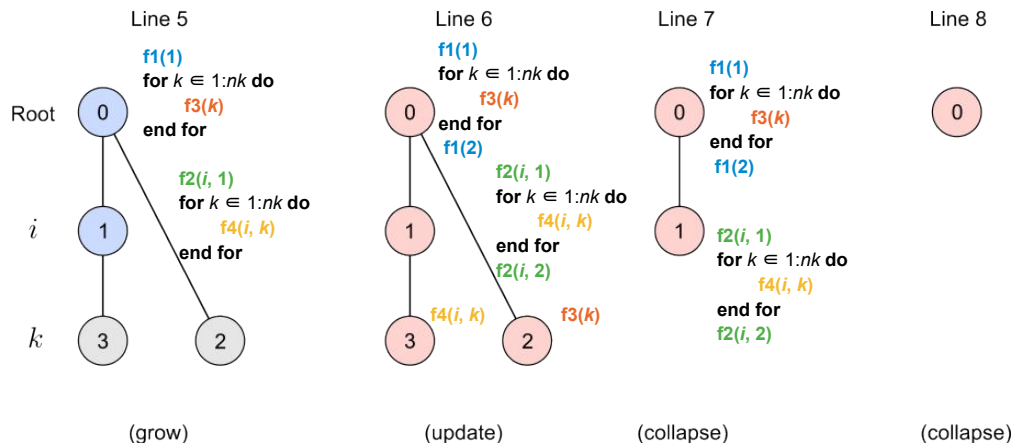
Dynamic binomial tree with  
**grow** and **collapse** operations

(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)

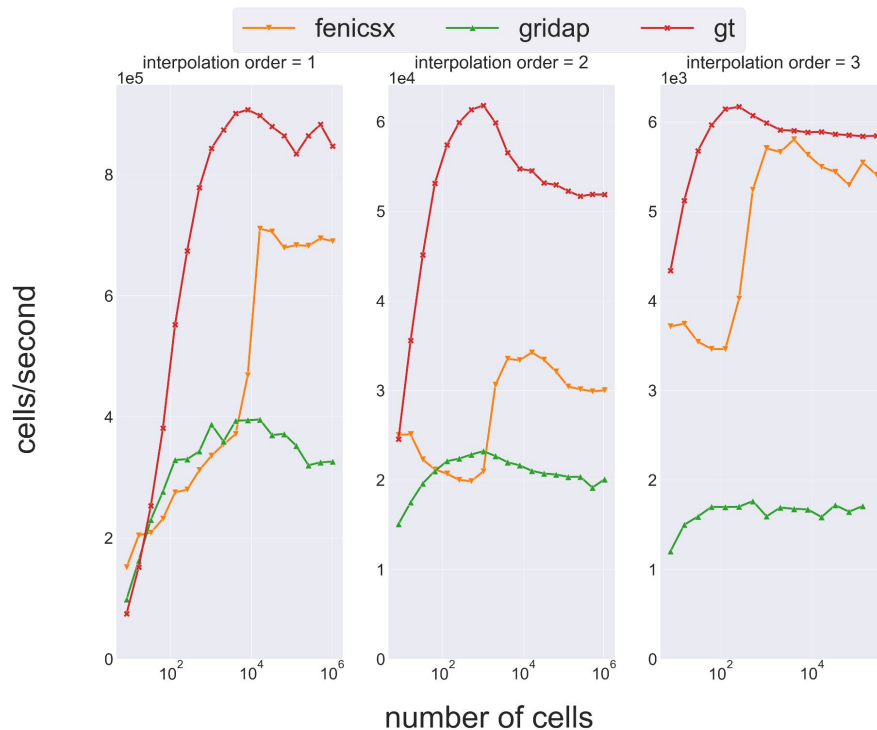
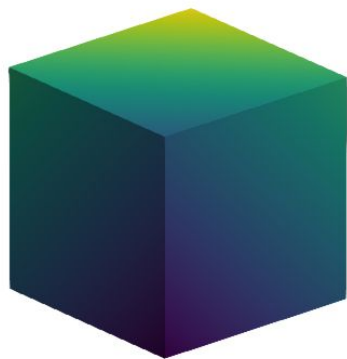
```
for i ∈ (1, ... ni) do
  for k ∈ (1, ... nk) do
    v += f1(1) * f2(i, 1) +
         f3(k) * f4(i, k)
  end for
  for k ∈ (1, ... nk) do
    v += f1(2) * f2(i, 2) +
         f3(k) * f4(i, k)
  end for
end for
```



```
... 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

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



# BENCHMARKING: POISSON EQUATION WITH DISCONTINUOUS GALERKIN

- The problem:

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

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

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

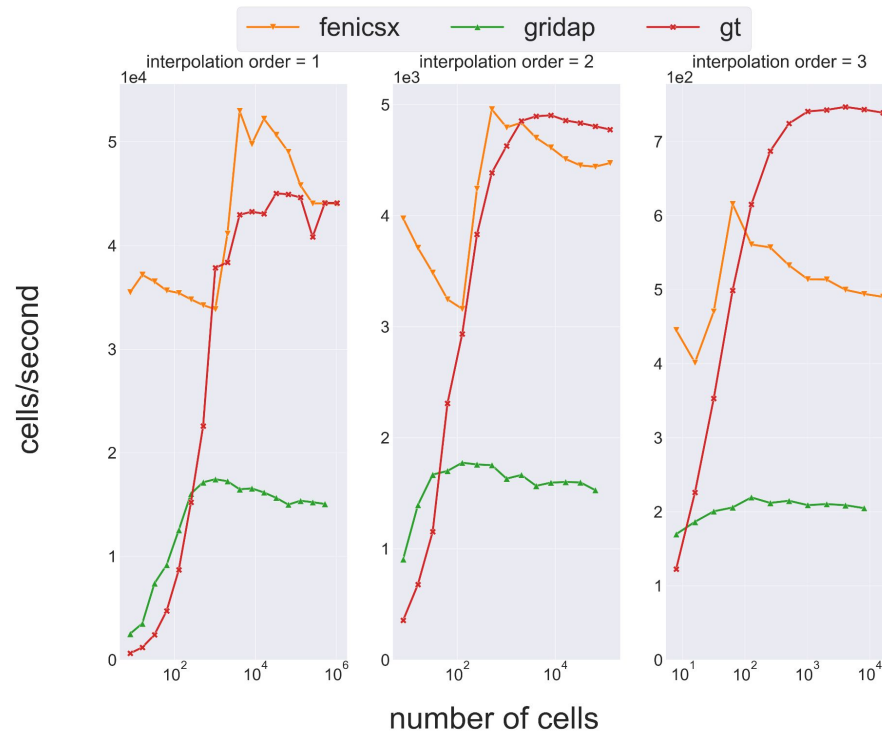


# BENCHMARKING: POISSON EQUATION WITH DISCONTINUOUS GALERKIN

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

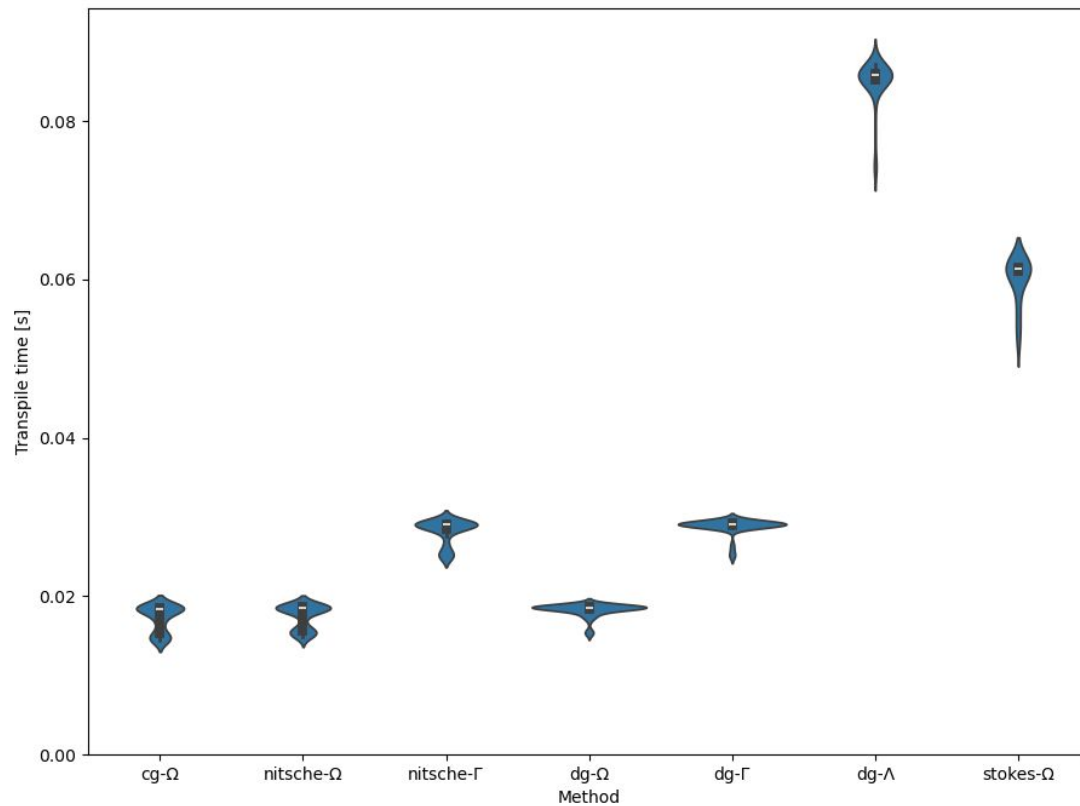
$$\int_{\Omega} \left( -\text{jump}(v, n) \cdot \text{mean}(\nabla u) - \text{jump}(u, n) \cdot \text{mean}(\nabla v) + \frac{\gamma}{h} \text{jump}(u$$

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



# 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 respect 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?