Imperial College London

Generating Optimised Multiplatform Finite Element Solvers from High-level Representations

Graham Markall[1], David Ham[2] and Paul H J Kelly[1]
Imperial College
http://www.doc.ic.ac.uk/~grm08
grm08@doc.ic.ac.uk

1. Software Performance Optimisation Group
2. Department of Earth Science & Engineering and Grantham Institute for Climate Change

Invitation to an Argument

Producing ***maintainable, high-performance*** implementations of the finite element method for multiple targets

REQUIRES

Writing them using a **high-level domain-specific language**

How to win the argument

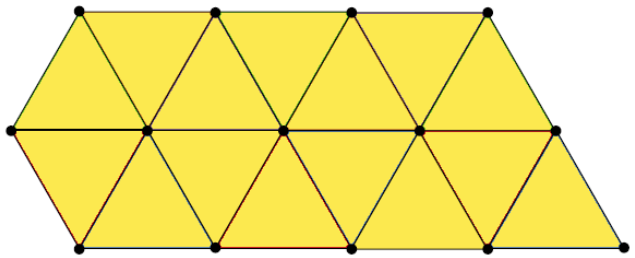
1. The Finite Element Method, Nvidia GPUs
2. High-performance FE solvers for
   - A multicore CPU
   - a Graphics Processing Unit (GPU)
   - - require different algorithms and data structures
3. The *Unified Form Language*
4. Code generation from UFL
5. UFL Bonus – Type checking
6. Ongoing work & conclusions

Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:

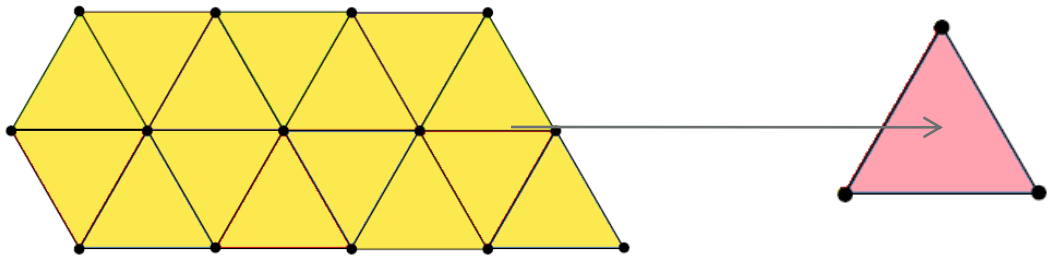


Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:



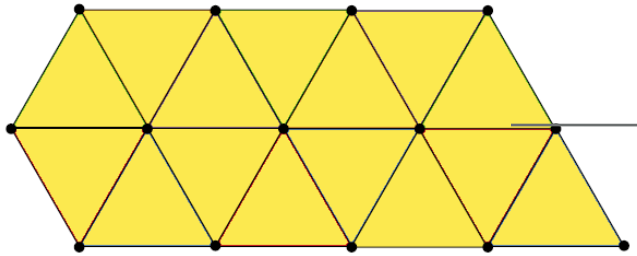
```
Do element =1,N
  Assemble(element)
End do
```

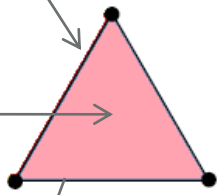
Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

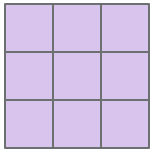
Domain:





```
Do element =1,N
  Assemble(element)
End do
```

Evaluate Integral
(Gaussian Quadrature)

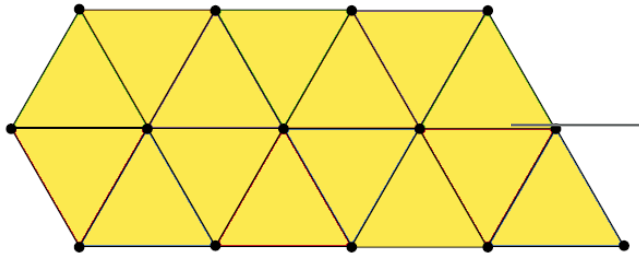


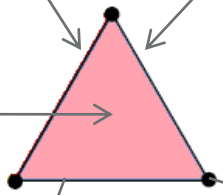
Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:

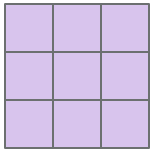


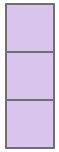


```
Do element =1,N
  Assemble(element)
End do
```

Evaluate q at nodes

Evaluate Integral
(Gaussian Quadrature)



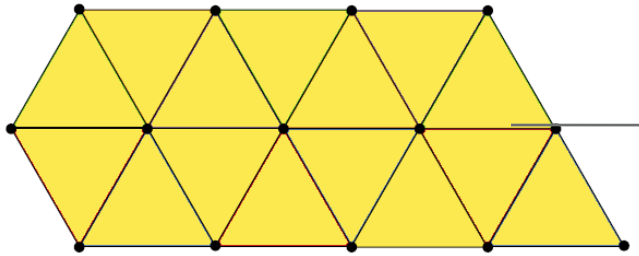


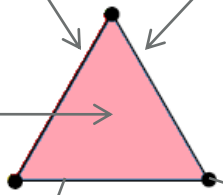
Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:

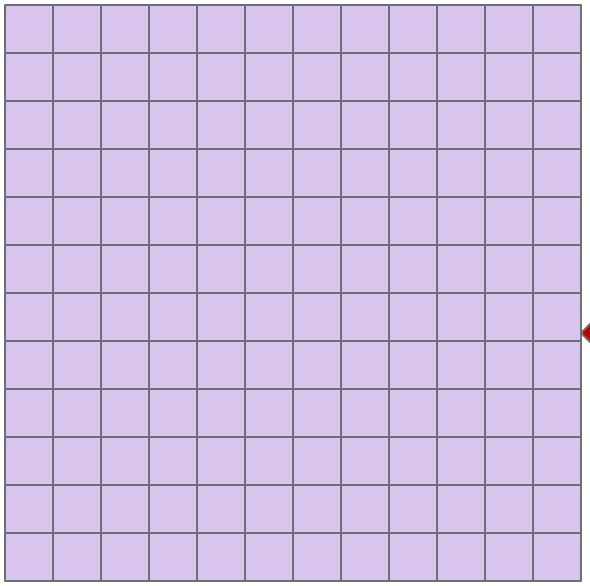


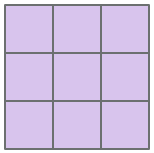


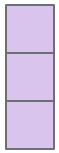
```
Do element =1,N
  Assemble(element)
End do
```

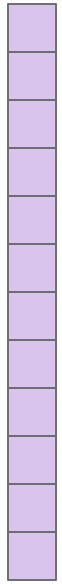
Evaluate q at nodes

Evaluate Integral
(Gaussian Quadrature)









Vector

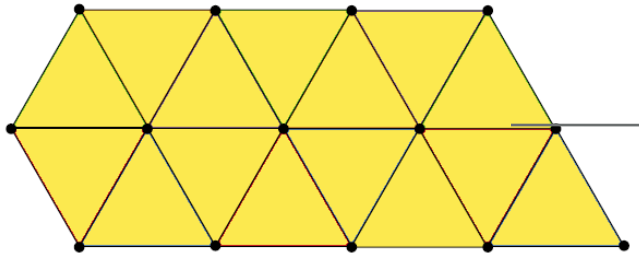
Compressed Sparse Row Matrix

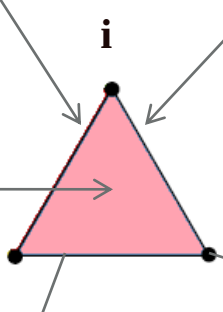
Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:

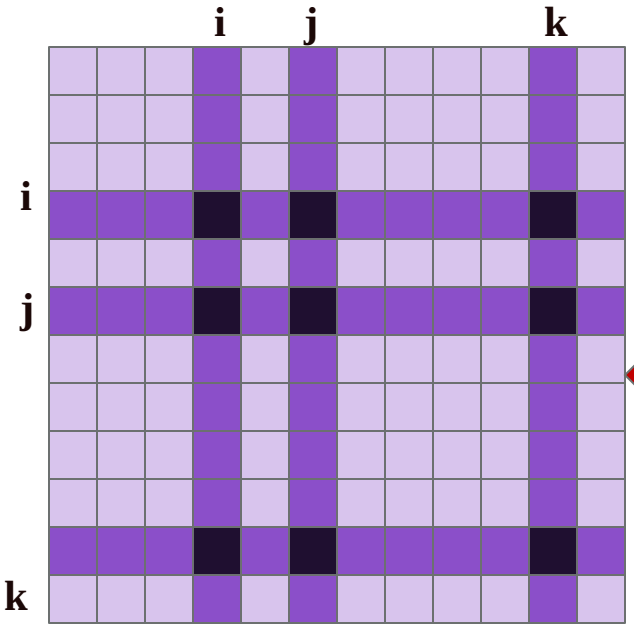


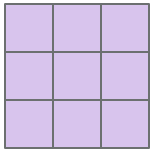


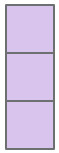

```
Do element =1,N
  Assemble(element)
End do
```

Evaluate q at nodes

Evaluate Integral (Gaussian Quadrature)







Vector

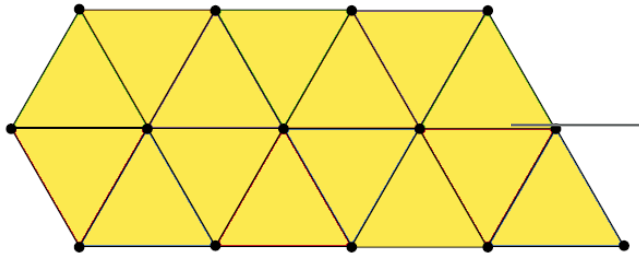
Compressed Sparse Row Matrix

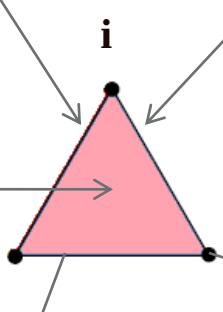
Imperial College London

A brief overview of the Finite Element Method

$$L(u) = q \longrightarrow \int_\Omega v L(u^\delta) \mathrm{d}X = \int_\Omega v q \mathrm{d}X.$$

Domain:

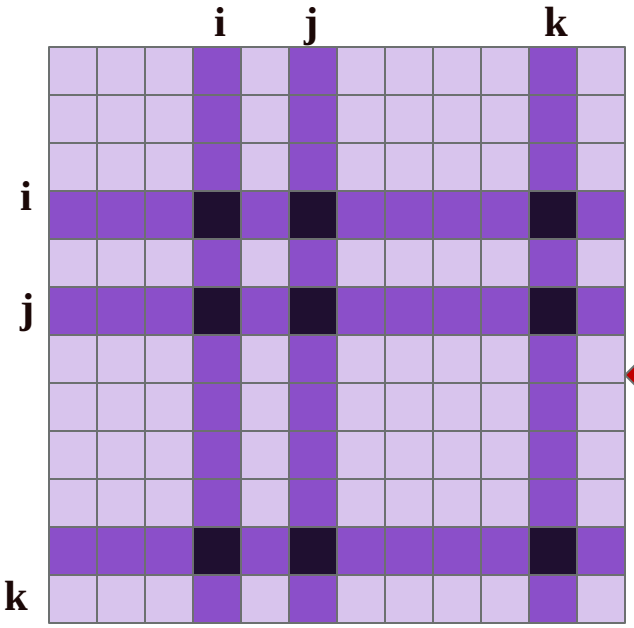


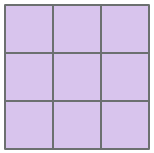


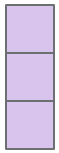

```
Do element =1,N
  Assemble(element)
End do
```

Evaluate q at nodes

Evaluate Integral (Gaussian Quadrature)







Vector

$$Mx = b$$

Solve for x

(CG, GMRES)

Compressed Sparse Row Matrix

Stages of the finite element method (per timestep)

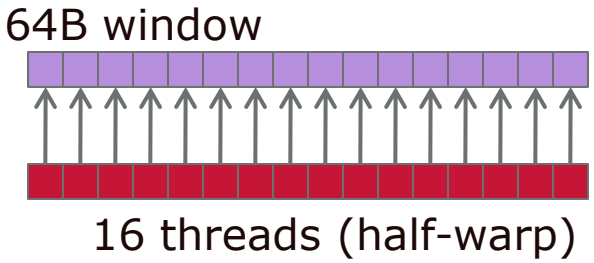
1. Local assembly:
   - Computation of M_e and b_e (Loop nest)

2. Global Assembly:
   - Addition of all the M_e into M, all the b_e into b

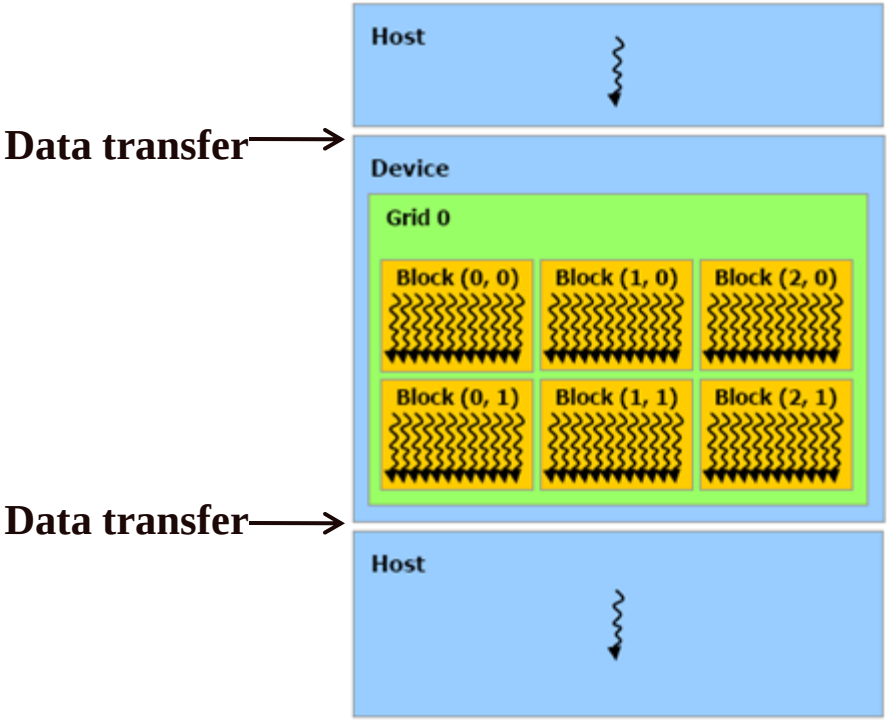
3. Solution
   - Iterative method, using sparse matrix-vector product Mv

NVIDIA Tesla & the CUDA Language

- **GT200 Architecture:**
  - 1-4GiB RAM
  - SIMT / MIMD / Vector
- **For high performance:**
  - Coalescing:

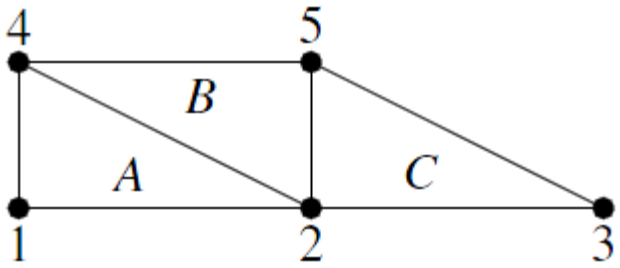


  - Use many threads (10000+)



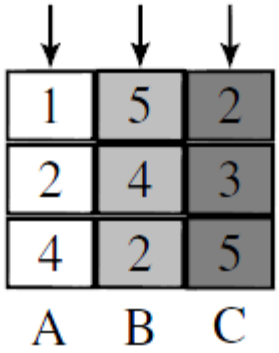
Local Assembly Implementation Issue – Data layout

Example 2D Domain:



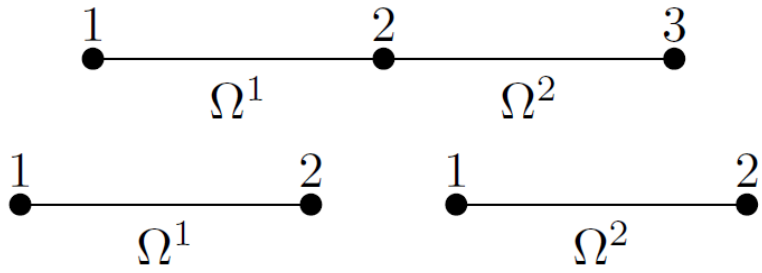
Data Layout in CPU implementation: (indirect access)

Data layout in GPU implementation: (allows *coalescing*)



Global Assembly Implementation options

Example 1D Domain:



$$M = A^T M^e A$$

$$b = A^T b^e$$

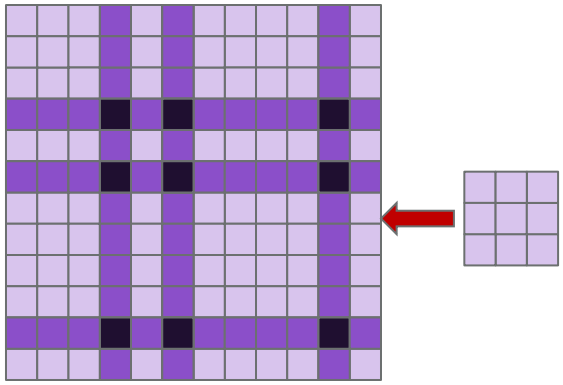
$$\mathcal{A} = \begin{bmatrix} 1 & & \\ & 1 & \\ & 1 & \\ & & 1 \end{bmatrix} \qquad \mathbf{b}^e = \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_1^2 \\ b_2^2 \end{bmatrix}$$

$$\mathbf{M}^e = \begin{bmatrix} m_{11}^1 & m_{12}^1 & & \\ m_{21}^1 & m_{22}^1 & & \\ & & m_{11}^2 & m_{12}^2 \\ & & m_{21}^2 & m_{22}^2 \end{bmatrix}$$

Imperial College London

Global Assembly Implementation options - 2

Option 1: the *Addto* algorithm:



Option 2: *the Local Matrix Approach*

$$y = Mv$$ (SpMV in solver)

$$y = \left(A^T\left(M^e(A\,v)\right)\right)$$

No computation of:
$M = A^T M^e A$

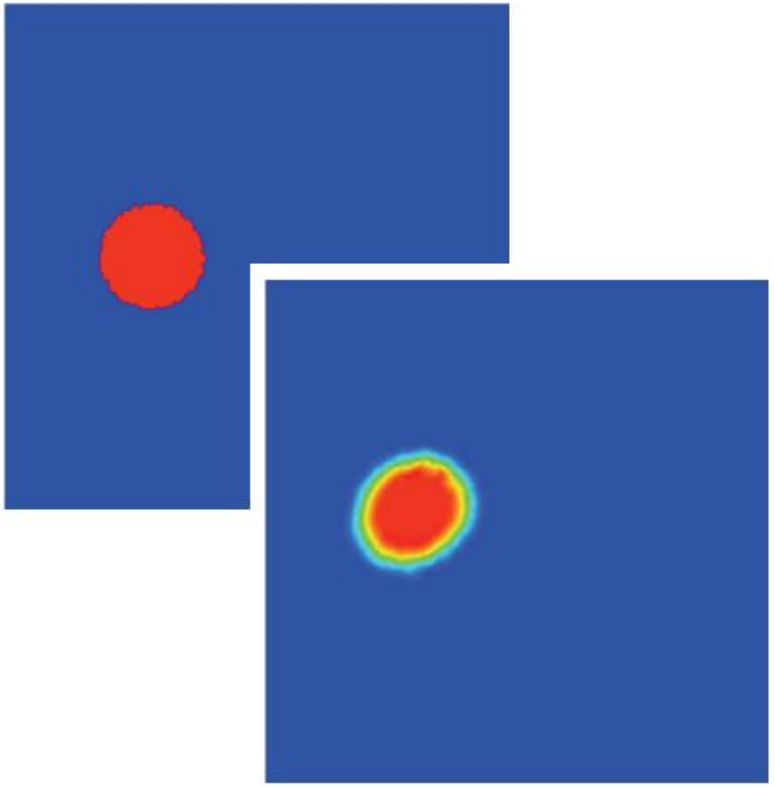
Right-hand side assembly: $b = A^T b^e$

Imperial College London

The test problem

Advection-diffusion eqn:

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \nabla \cdot \overline{\overline{\mu}} \cdot \nabla T$$



Simplified model and sub-problem of full fluid dynamics solver, e.g.

- Ocean model
- Turbine model

Implemented within Fluidity:

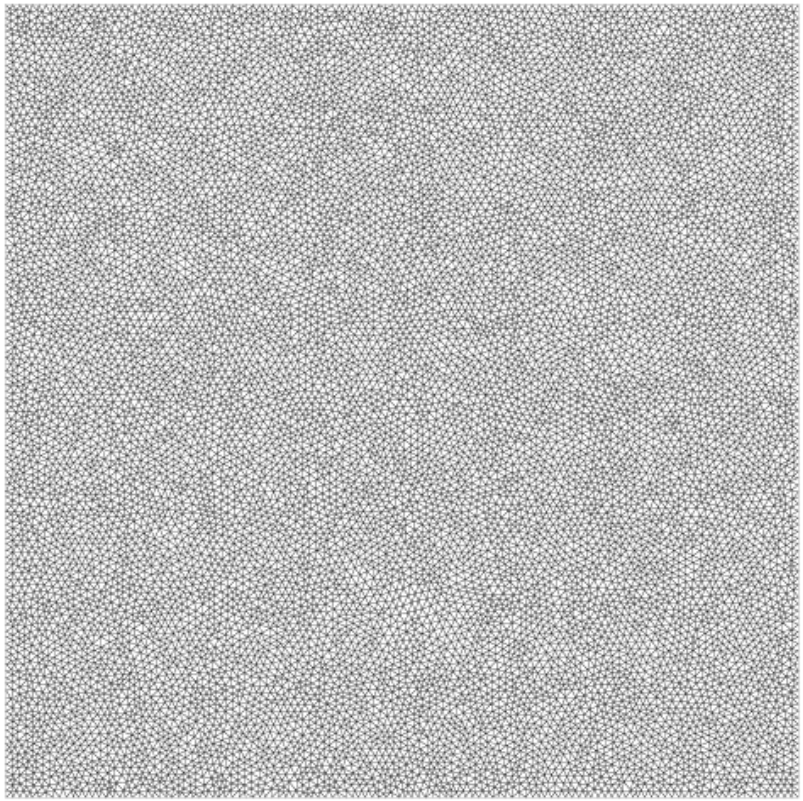
- General-purpose CFD code with real users

Imperial College London

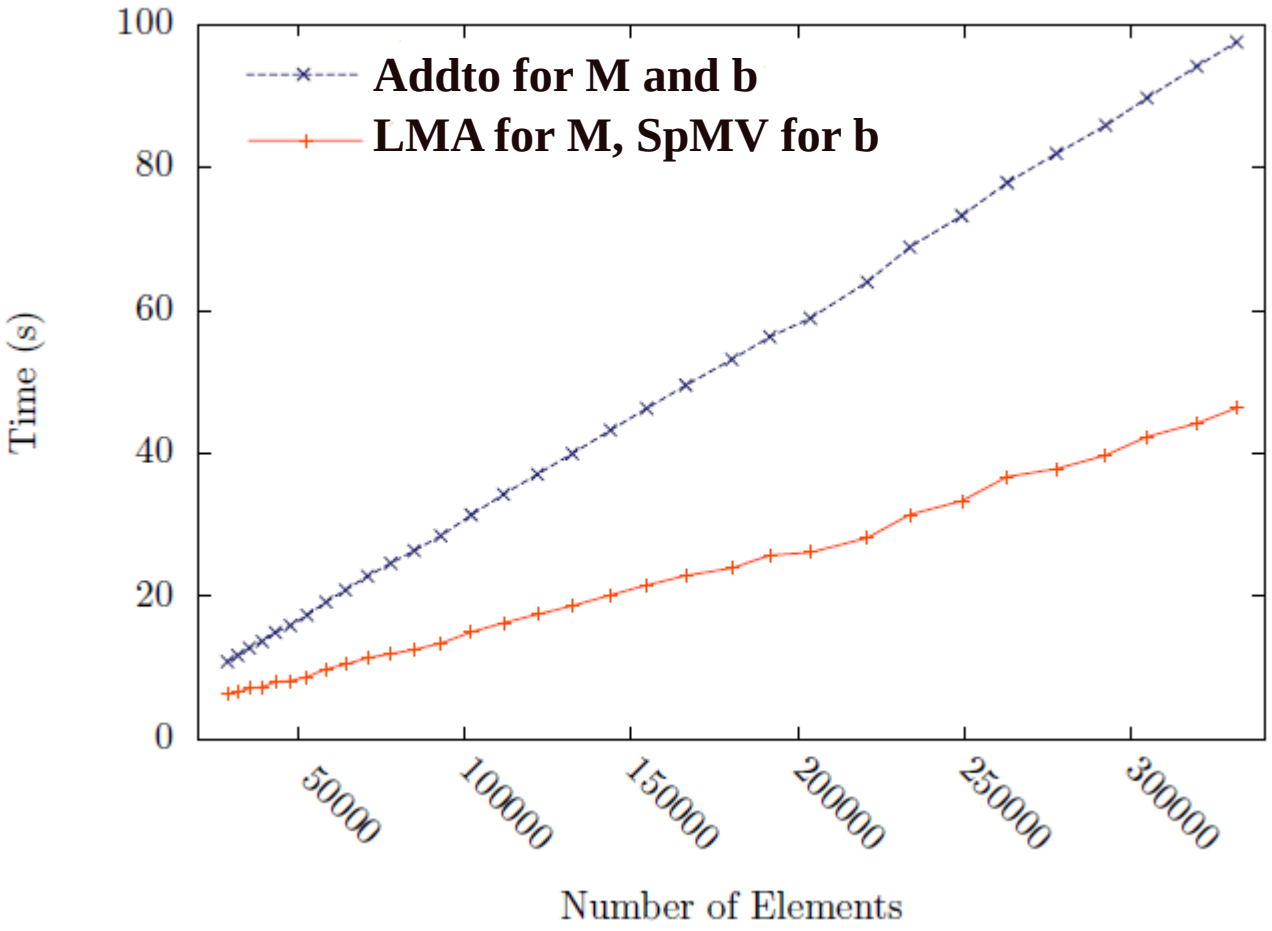
Performance evaluation setup

- Nvidia 280GTX – 1GB RAM (use Tesla C1060 for 4GB)
- Intel Core 2 Duo E8400 @ 3.00GHz
- Double Precision arithmetic
- Run problem for 200 timesteps

- 2GB RAM in host machine
- Intel C++ and Fortran Compilers V10.1 - V11.0 suffers from bugs and cannot compile Fluidity
- Five runs of each problem - averages reported
- CPU Implementations compiled with **–O3** flags
- CUDA Implementation compiled using NVCC 2.2
- Increasingly finer meshes with increasing element count



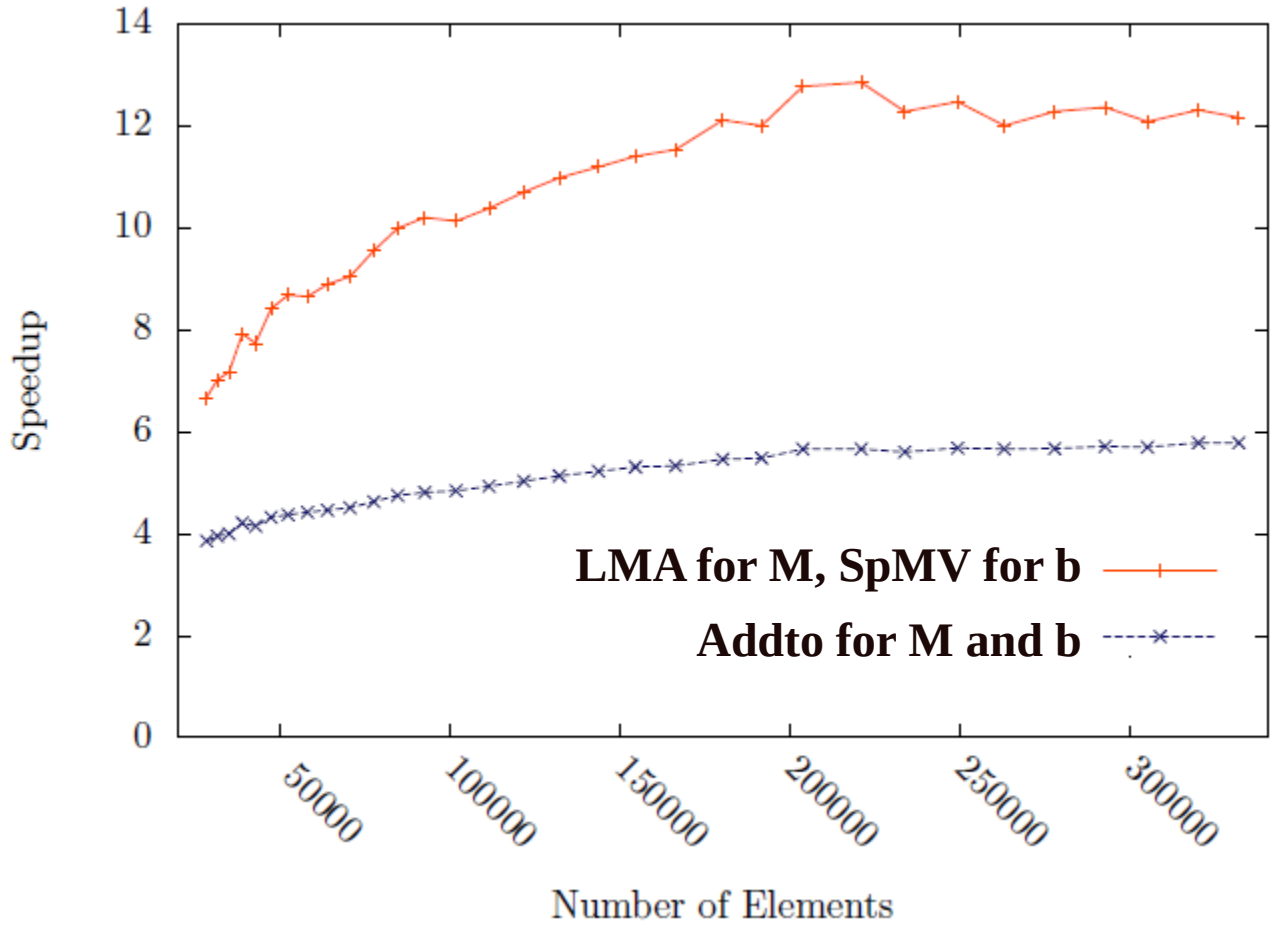
GPU Performance results



Global Assembly on GPU: The Local Matrix Approach is the optimal approach

Imperial College
London

Speedup relative to CPU Implementation



CPU implementation uses addto algorithm: More efficient than LMA for low-order elements. [Vos & Sherwin 2010]

Results Summary

- **High performance from GPUs**: Multiply-targeted finite element solvers worth pursuing
- **Data Layouts:** Different layouts for different targets (introduces a marshalling problem)
- **Algorithmic choice**: Necessary for optimal performance on each target
- **Motivate** the use of a high-level DSL: the ***Unified Form Language*** [Logg, 2006]

Imperial College London

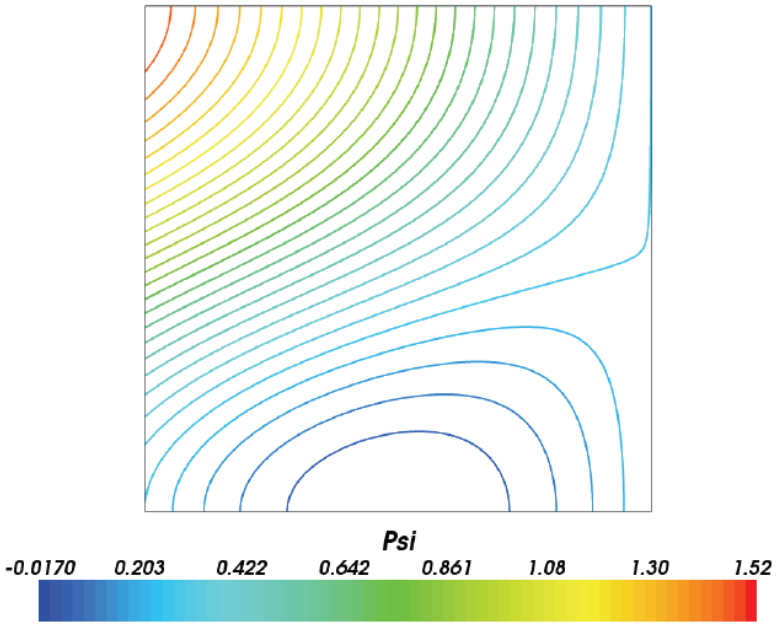
UFL Example – Poisson's Equation

$$\nabla^2 u = f$$

$$\int_\Omega \nabla v \cdot \nabla u \, dX = \int_\Omega v f \, dX$$

(Ignoring boundary conditions)

```
P=state.scalar_fields("Pressure")
v=TestFunction(P)
u=TrialFunction(P)
f=Function(P)
A=dot(grad(v),grad(u))*dx
RHS=v*f*dx
Pressure=Solve(A,RHS)
```



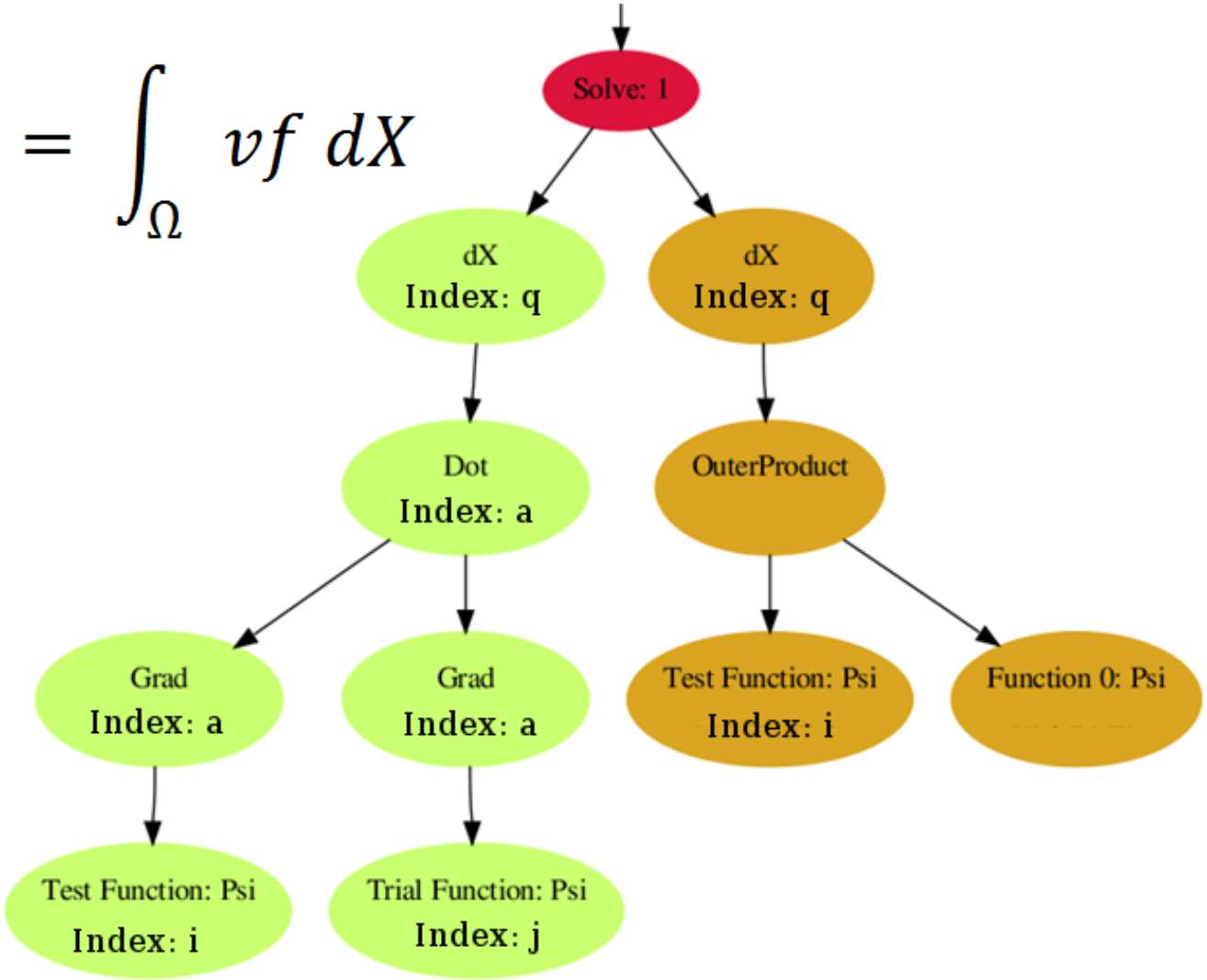
c.f. a typical Fortran implementation using a domain library, ~500 LOC

The Unified Form Language

- From the FEniCS project
- Desirable to use:
  - Maths-like notation
  - Independent of the underlying implementation
  - Reduce LOC by an order of magnitude
- Code generation possibilities:
  - Targets
  - Data layouts
  - Algorithms
  - Marshalling code

An Abstract Intermediate Representation

$$\int_\Omega \nabla v \cdot \nabla u \, dX = \int_\Omega v f \, dX$$



Code Generation - Preliminaries

To compute: $\int_{\Omega} \nabla v \cdot \nabla u \, dX$

For each element *E*, compute a matrix *M*:

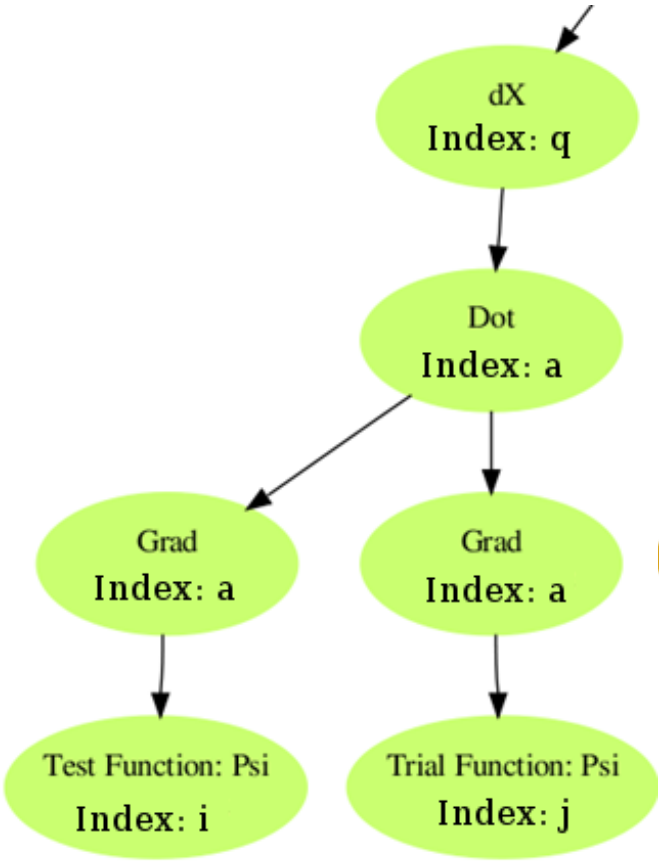
$$M[i,j] = \int_{E} \nabla \phi_i \cdot \nabla \psi_j dX \quad i = 1 \ldots N_{\phi}, j = 1 \ldots N_{\psi}$$

Evaluating integrals using Gaussian quadrature:
(implementation choice)

$$\int_{E} \nabla \phi_i \cdot \nabla \psi_j dX = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

Imperial College
London

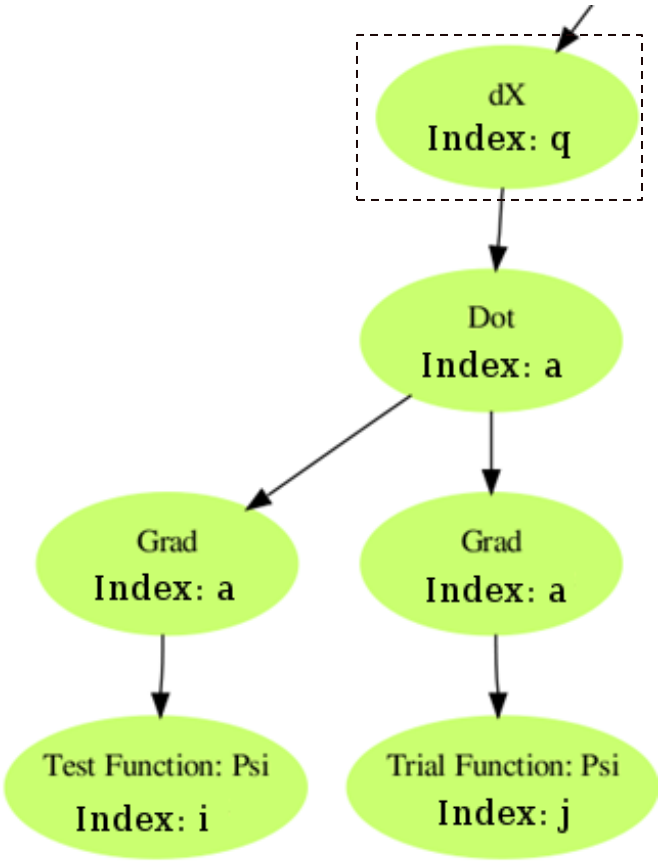
Code Generation & Optimisation – Generating Expressions



$$\sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

Imperial College London

Code Generation & Optimisation – Generating Expressions

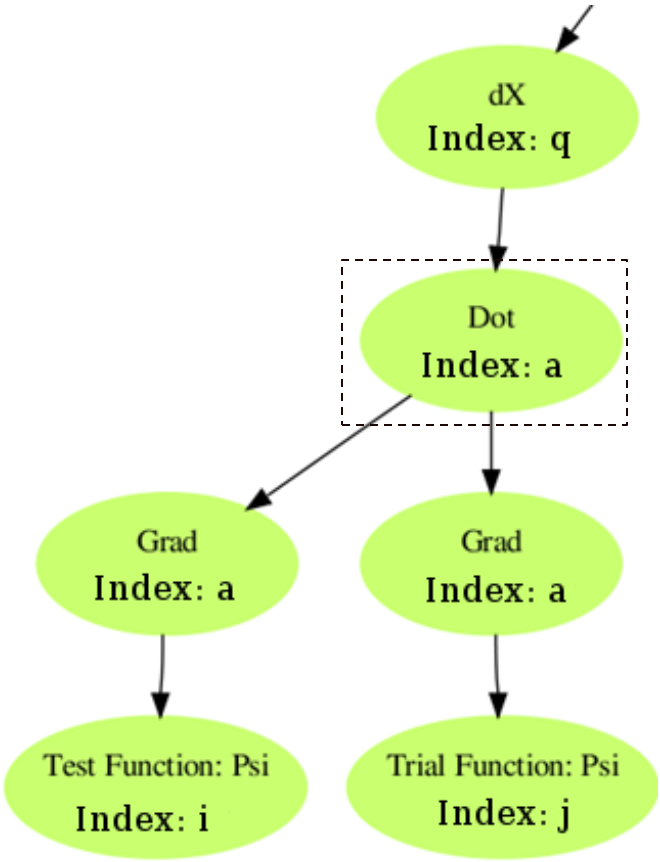


$$\sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
M[i,j] += wqJ[q]
          *
```

Imperial College London

Code Generation & Optimisation – Generating Expressions

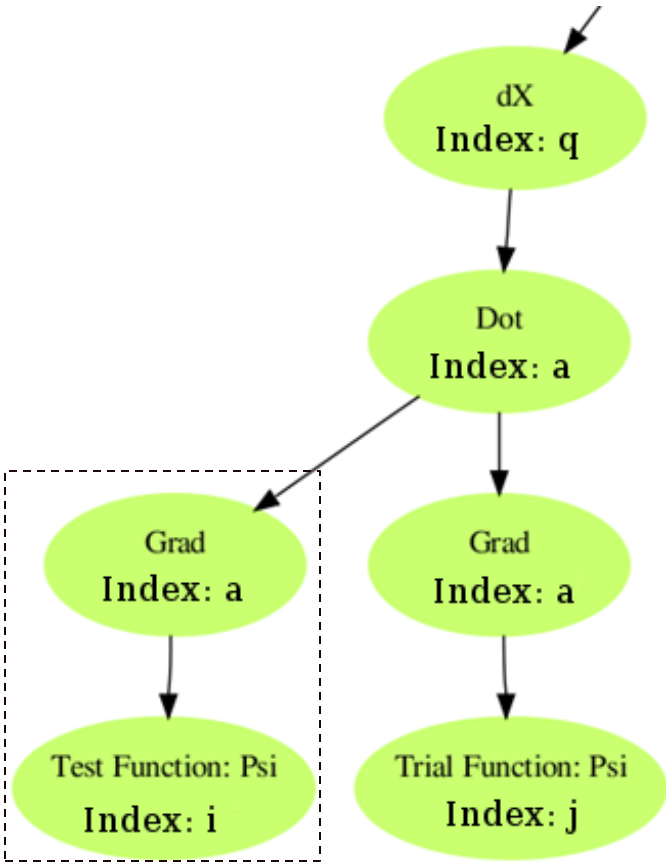


$$\sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
M[i,j] += wqJ[q]
          * (
          *                                  )
```

Imperial College London

Code Generation & Optimisation – Generating Expressions

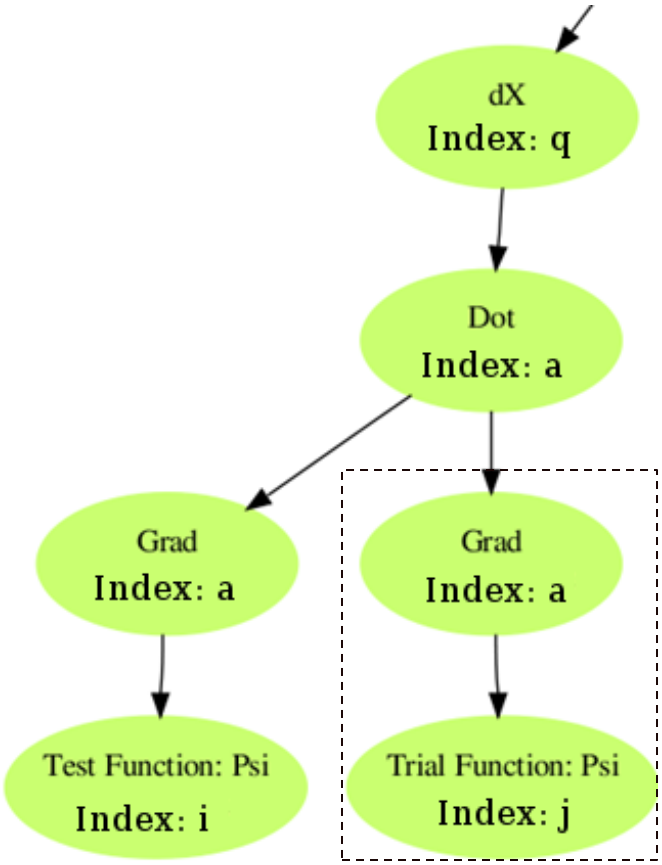


$$\sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
M[i,j] += wqJ[q]
          * (d_test_u[q,i,alpha]
          *                    )
```

Imperial College London

Code Generation & Optimisation – Generating Expressions

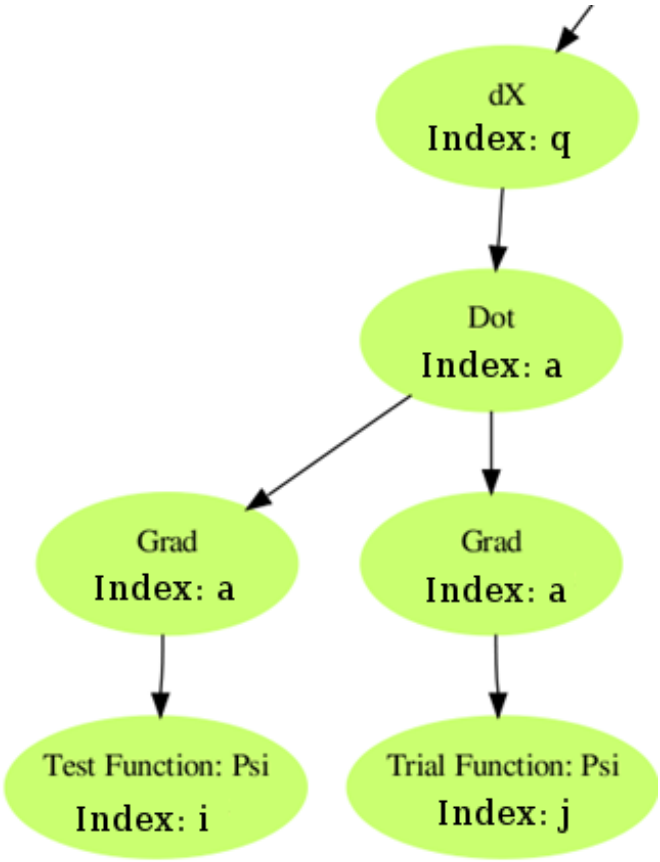


$$\sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

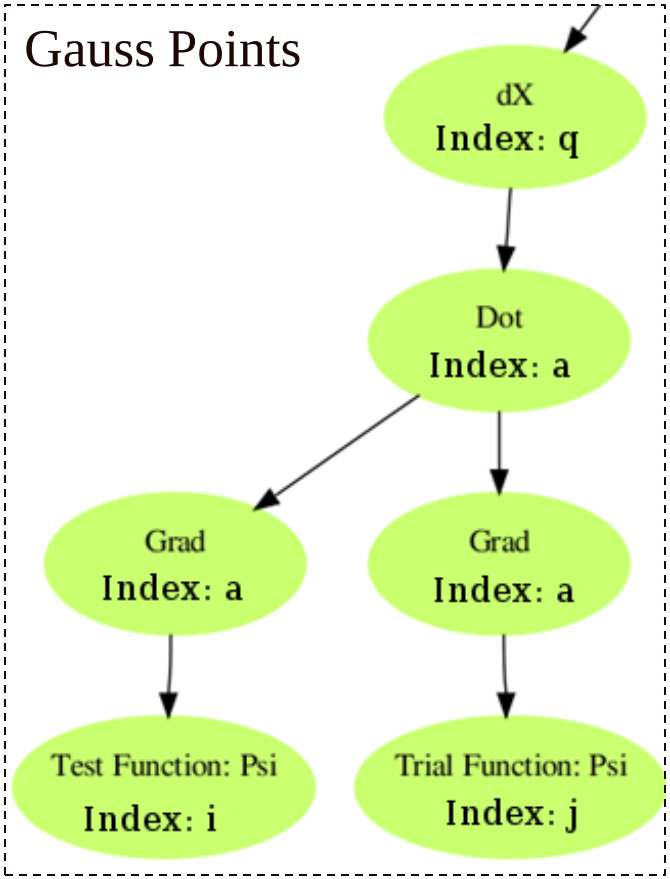
```
M[i,j] += wqJ[q]
        * (d_test_u[q,i,alpha]
        * d_trial_u[q,j,alpha])
```

Imperial College
London

Code Generation & Optimisation - Lowering to loops



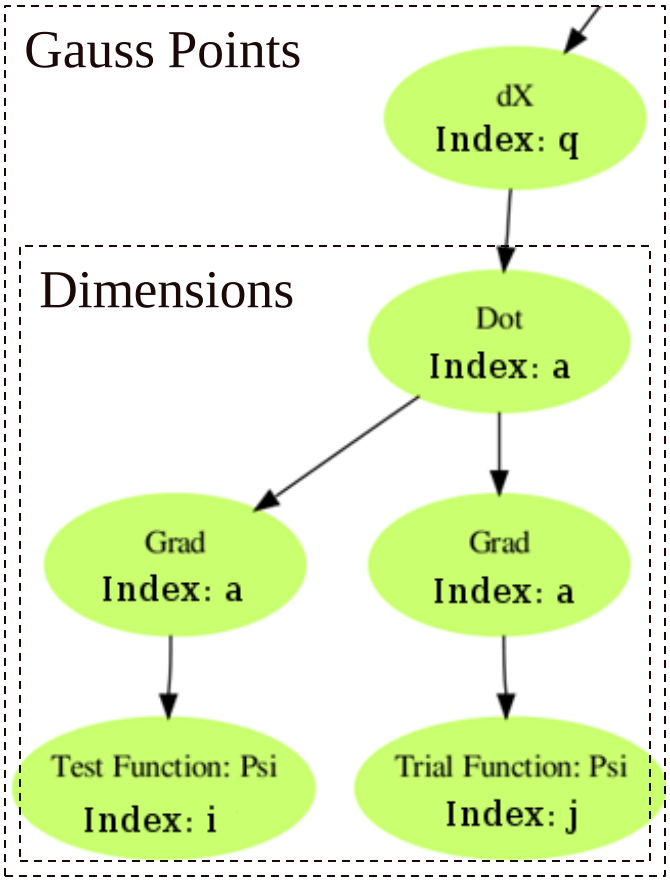
Code Generation & Optimisation - Lowering to loops




```
for q=1 to n_gauss_points
```

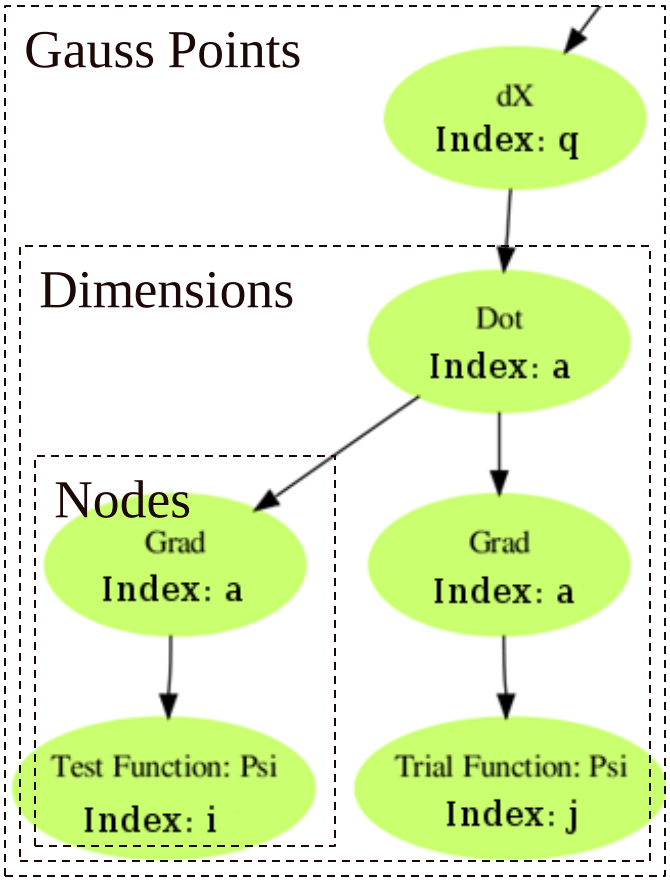
Imperial College London

Code Generation & Optimisation - Lowering to loops



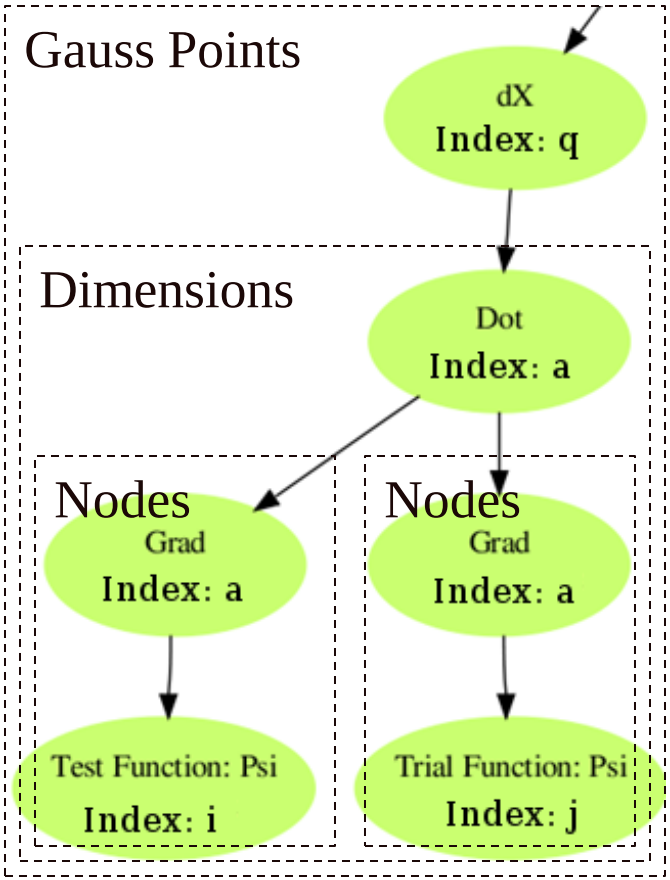

```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
```

Code Generation & Optimisation - Lowering to loops




```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
```

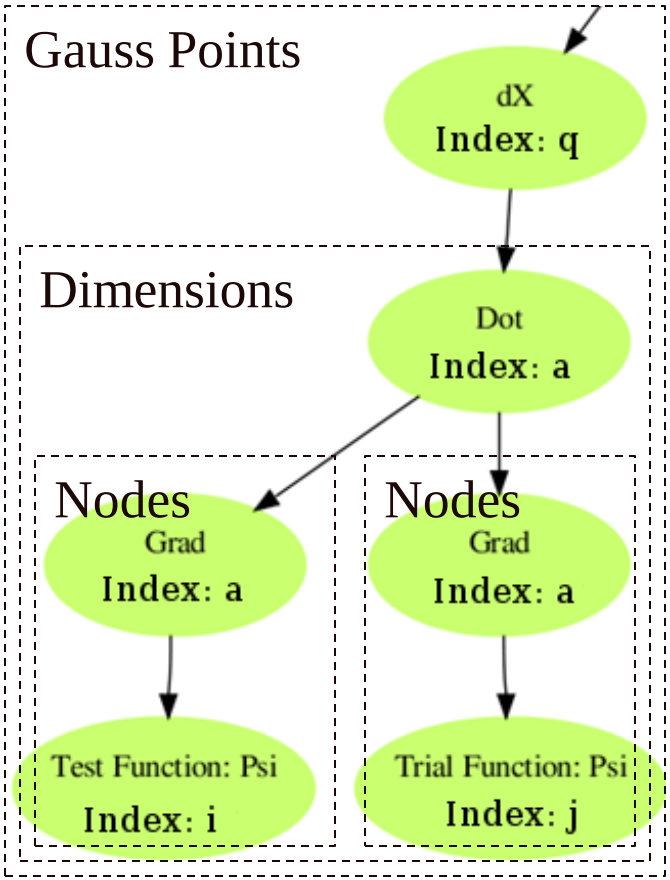
Code Generation & Optimisation - Lowering to loops




```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

Imperial College London

Code Generation & Optimisation - Lowering to loops

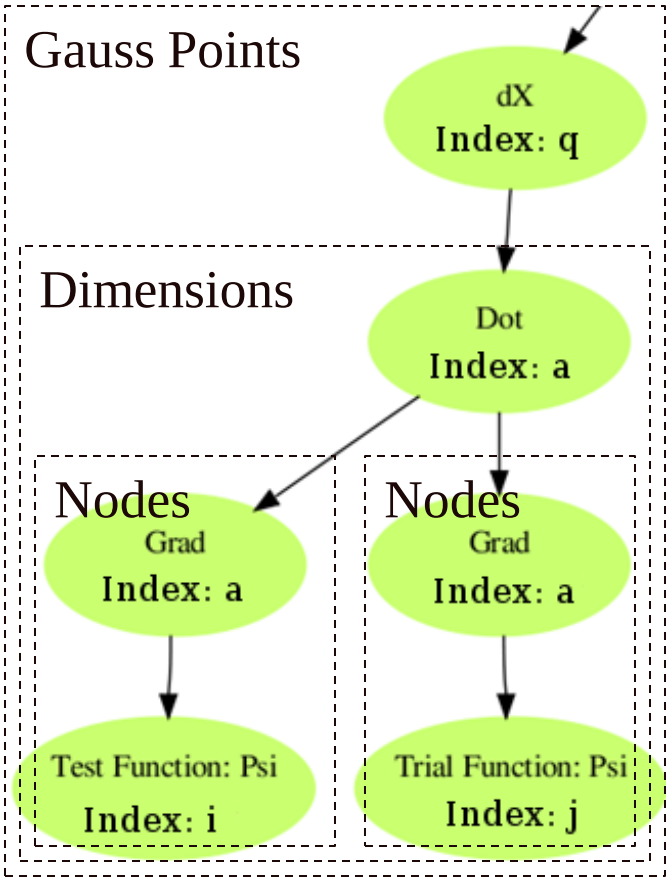



```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

$$M[i,j] = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

Imperial College London

Code Generation & Optimisation - Lowering to loops



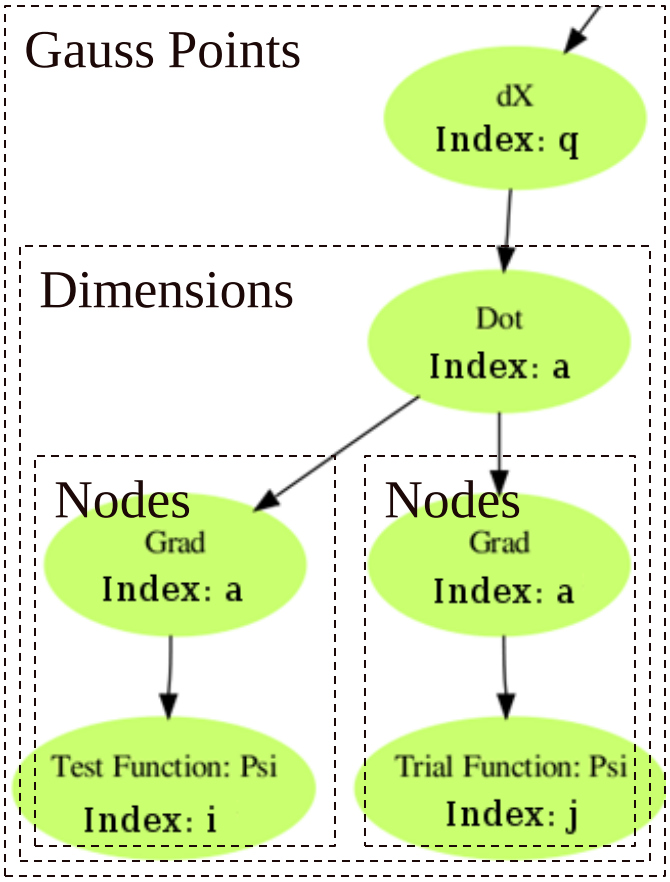

```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

$$M[i,j] = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
for i=1 to nodes_per_element
  for j=1 to nodes_per_element
```

Imperial College London

Code Generation & Optimisation - Lowering to loops



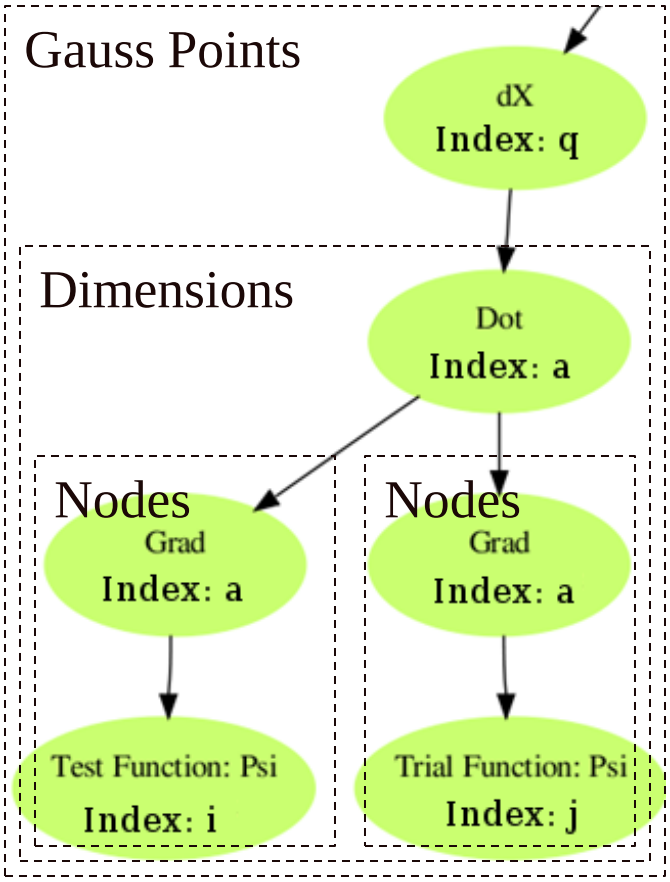

```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

$$M[i,j] = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
for i=1 to nodes_per_element
  for j=1 to nodes_per_element
    for q=1 to n_gauss_points
```

Imperial College London

Code Generation & Optimisation - Lowering to loops



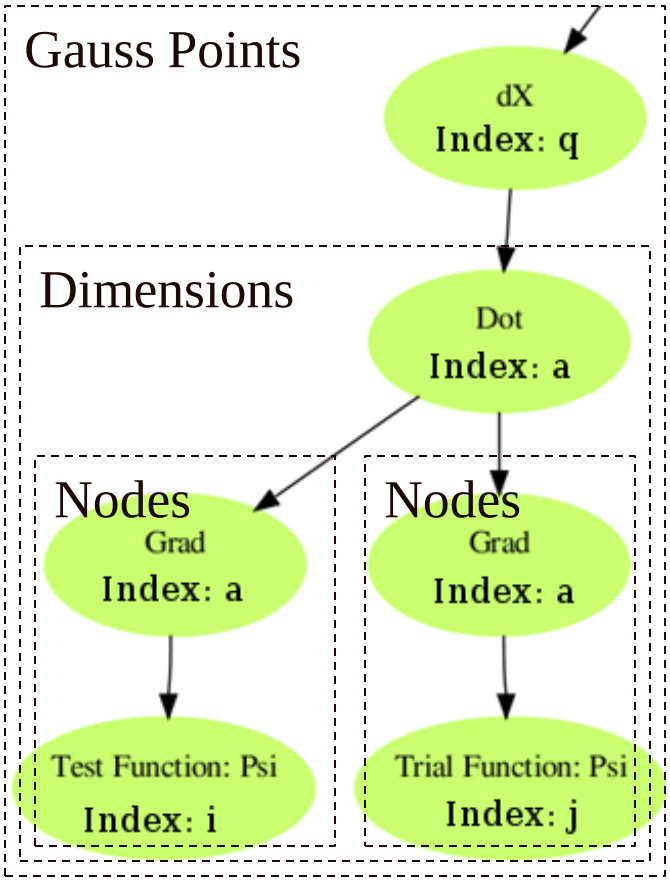

```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

$$M[i,j] = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
for i=1 to nodes_per_element
  for j=1 to nodes_per_element
    for q=1 to n_gauss_points
      for alpha=1 to n_dimensions
```

Imperial College London

Code Generation & Optimisation - Lowering to loops

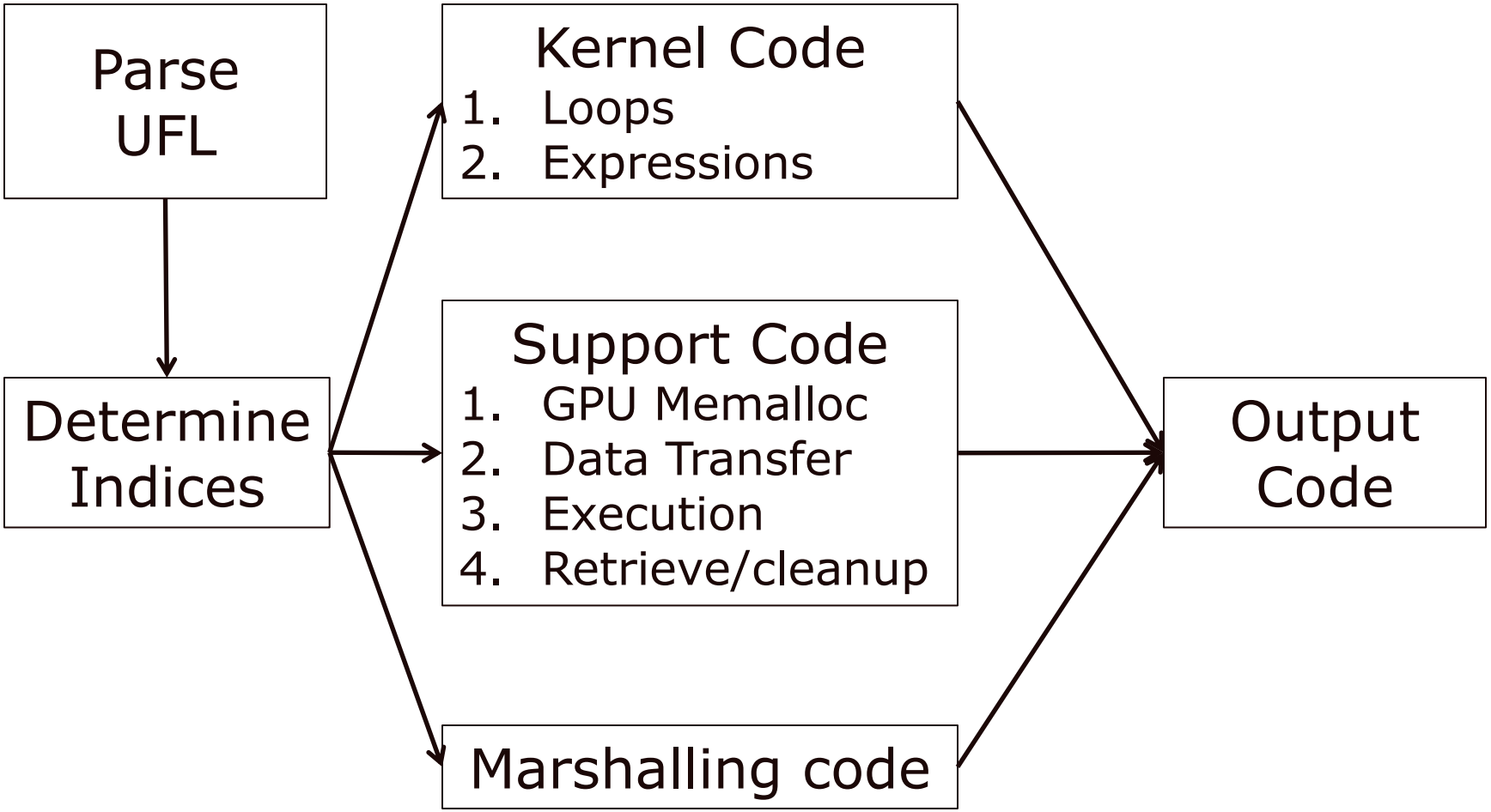



```
for q=1 to n_gauss_points
for alpha=1 to n_dimensions
for i=1 to nodes_per_element
for j=1 to nodes_per_element
```

$$M[i,j] = \sum_{q=1}^{N_q} \sum_{\alpha=1}^{N_{dim}} w_q \frac{\partial \phi_i}{\partial X_\alpha}(\xi_q) \frac{\partial \psi_j}{\partial X_\alpha}(\xi_q) J^e$$

```
for i=1 to nodes_per_element
  for j=1 to nodes_per_element
    for q=1 to n_gauss_points
      for alpha=1 to n_dimensions
        M[i,j] = ...
```

Code Generation - Summary



Imperial College London

UFL vs. generated CUDA code

```
rhs=dt*(dot(grad(q),u)*t-q*t*div(u))*dx
```

```
__global__ void adv_rhs_1(int dim, int loc, int ngi, int ele, double *M,
                          double *detwei, double *d_test_Tracer,
                          double *div_q_Velocity_2, double *val_q_Tracer_0,
                          double *test_Tracer)
{
 for (int i_ele = THREAD_ID; i_ele < ele; i_ele += THREAD_COUNT) {
    for (int i_4 = 0; i_4 < loc; ++i_4) {
      M[i_ele * loc + i_4] = 0;
      for (int i_1 = 0; i_1 < ngi; ++i_1) {
        M[i_ele * loc + i_4] += -1.00000 * (div_q_Velocity_2[i_ele * ngi + i_1]
                              * (test_Tracer[i_ele * (loc * ngi) + i_4 * ngi + i_1]
                              * val_q_Tracer_0[i_ele * ngi + i_1]))
                              * detwei[ngi * i_ele + i_1];
        for (int i_6 = 0; i_6 < dim; ++i_6) {
          M[i_ele * loc + i_4] += d_test_Tracer[i_ele * (dim * (loc * ngi))
                                                + i_6 * (loc * ngi) + i_4 * ngi + i_1]
                                * val_q_Velocity_2[i_ele * (dim * ngi) + i_6 * ngi + i_1]
                                * val_q_Tracer_0[i_ele * ngi + i_1]
                                * detwei[ngi * i_ele + i_1];
        }
      }
    }
  }
}
```

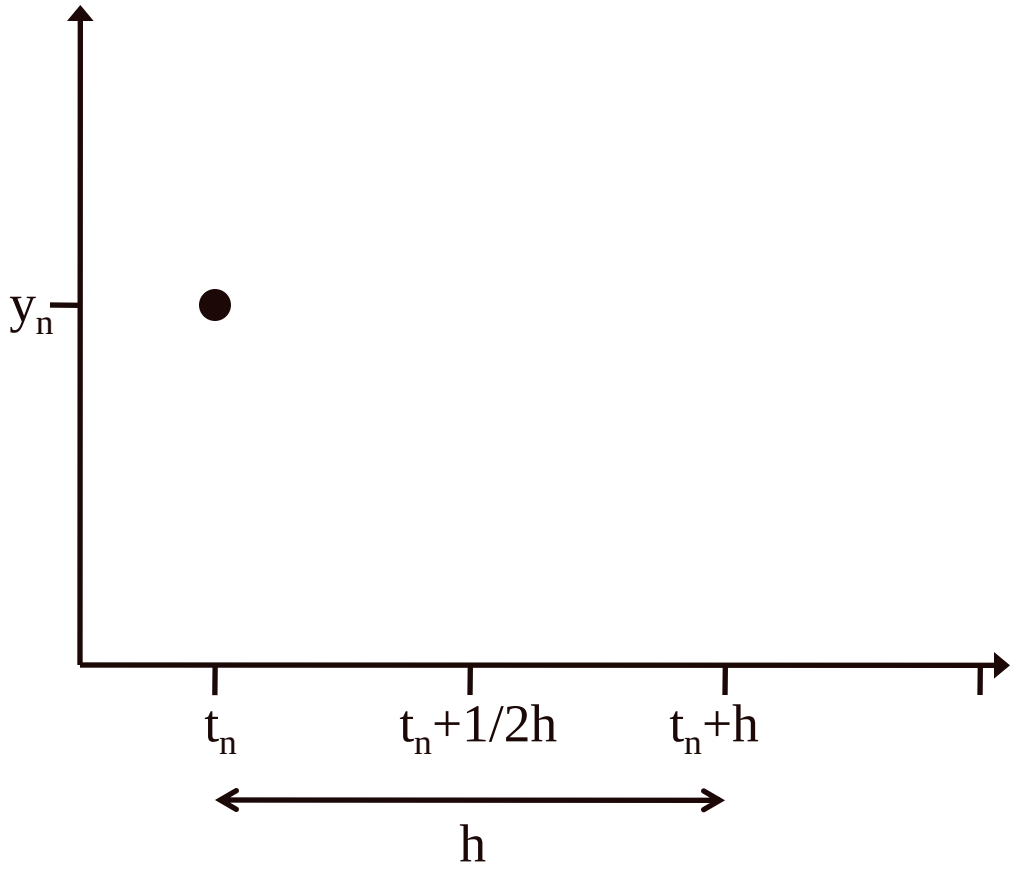
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



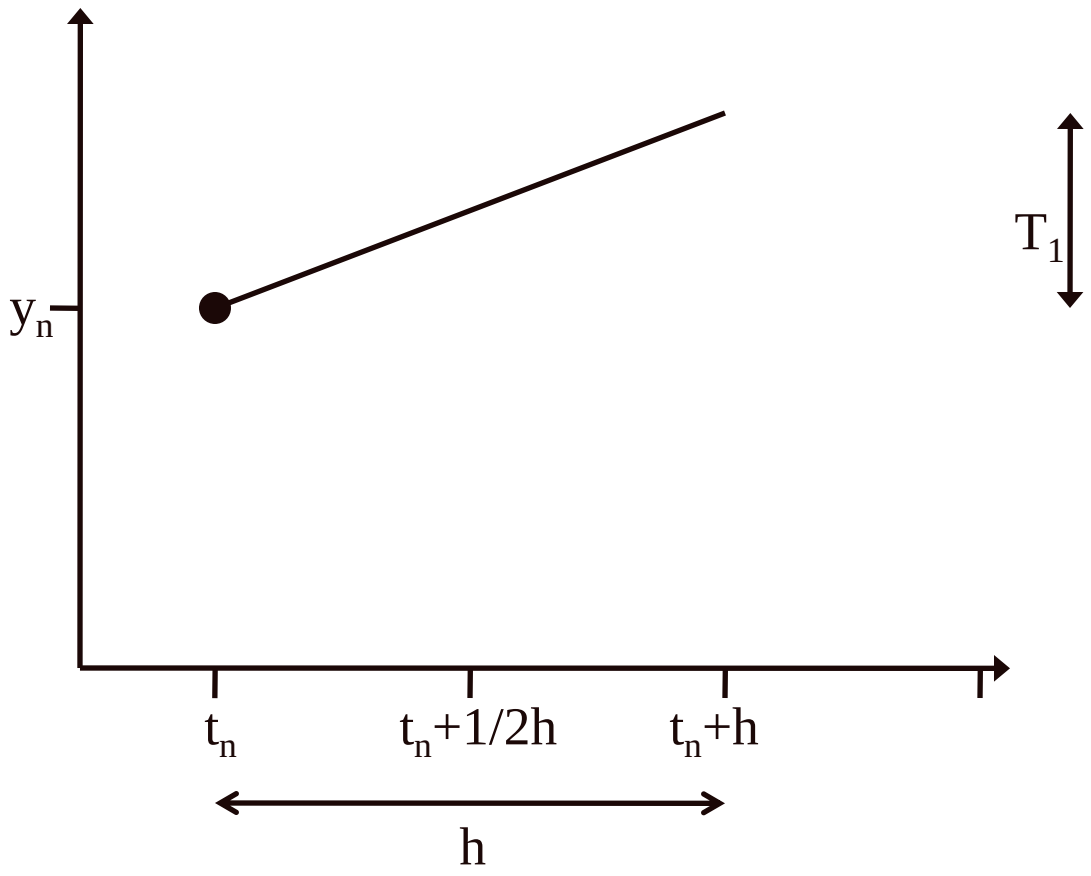
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



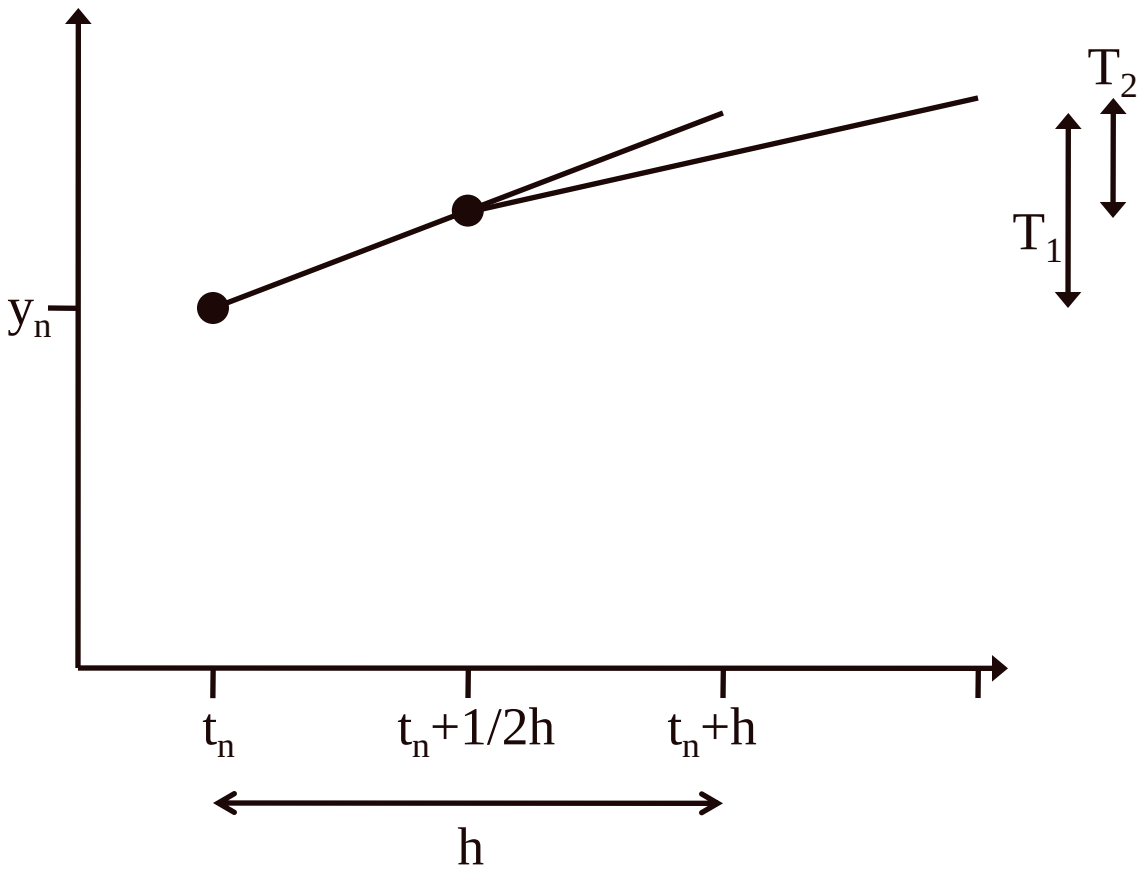
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



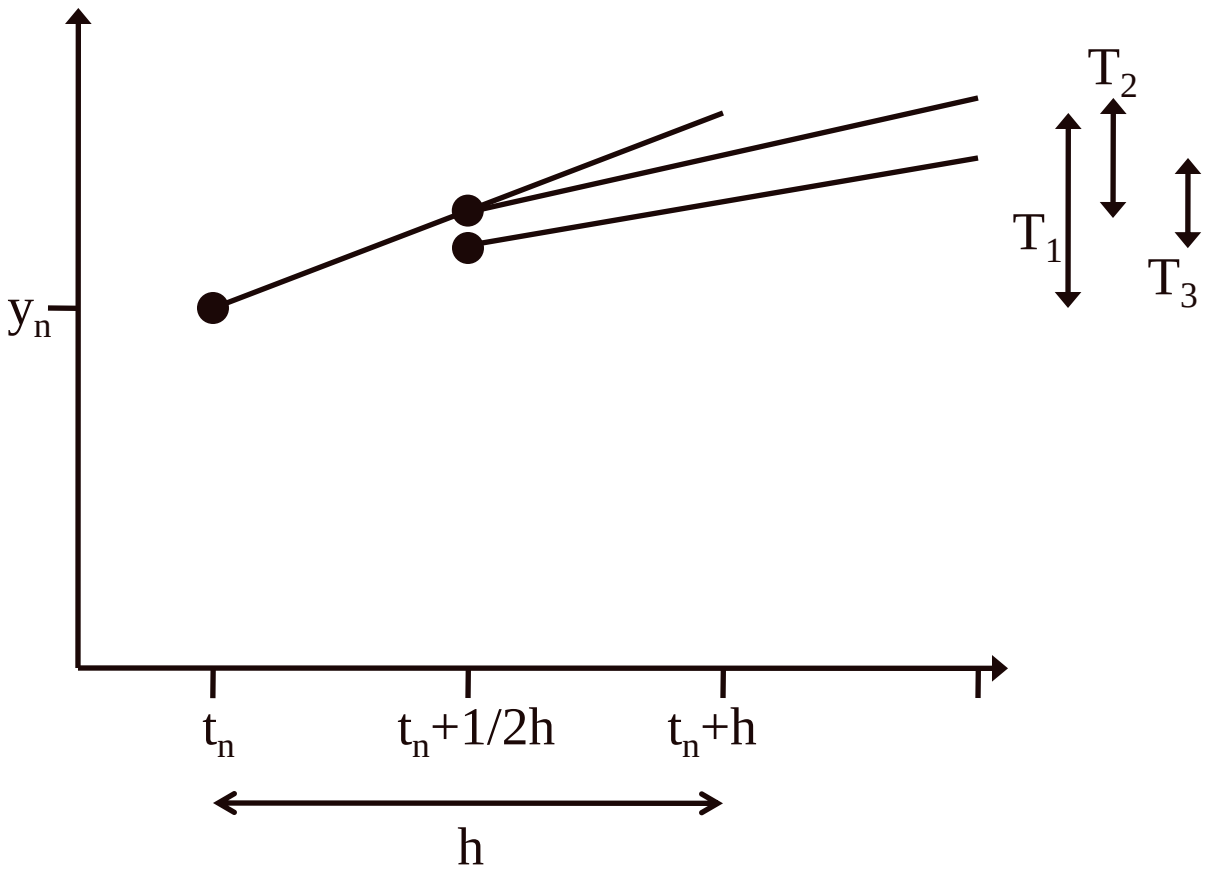
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



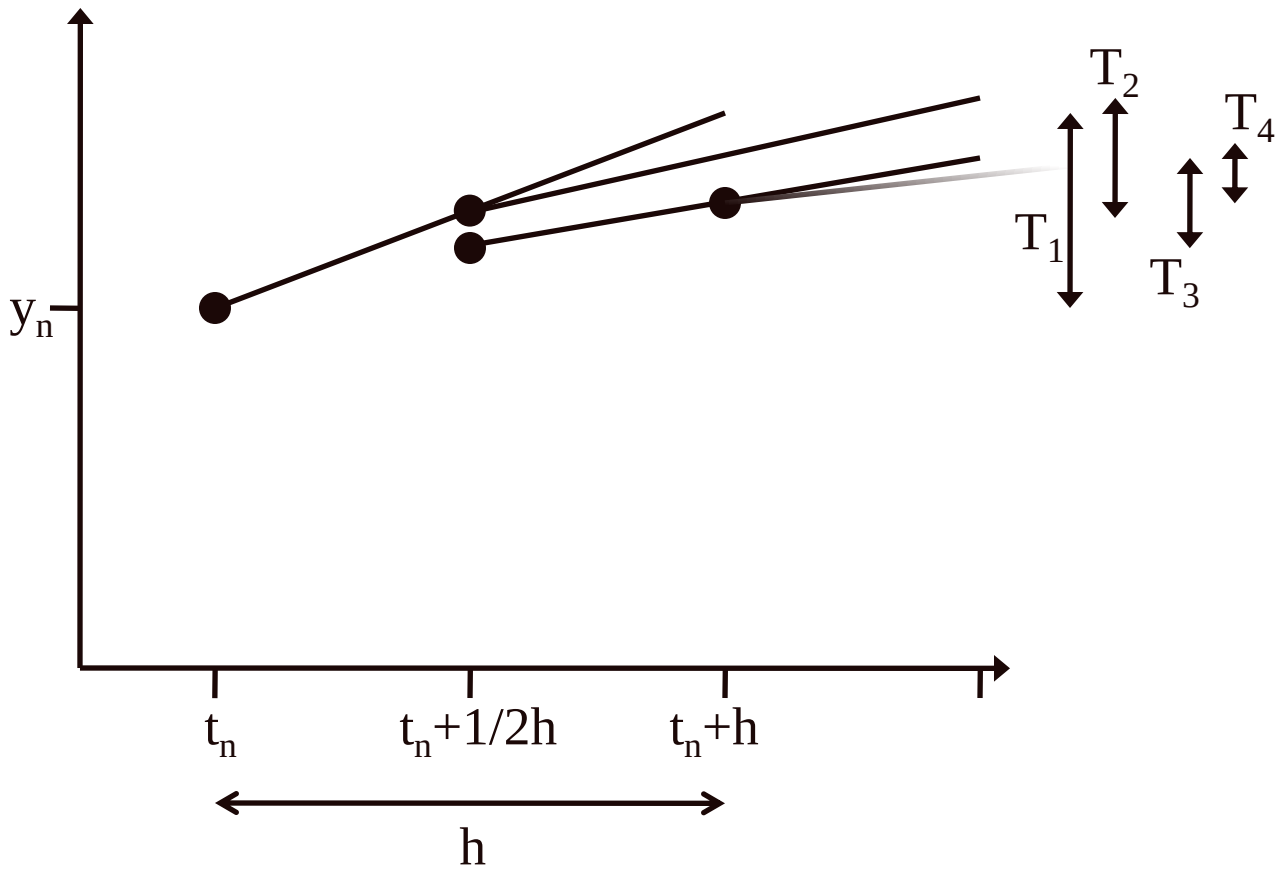
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



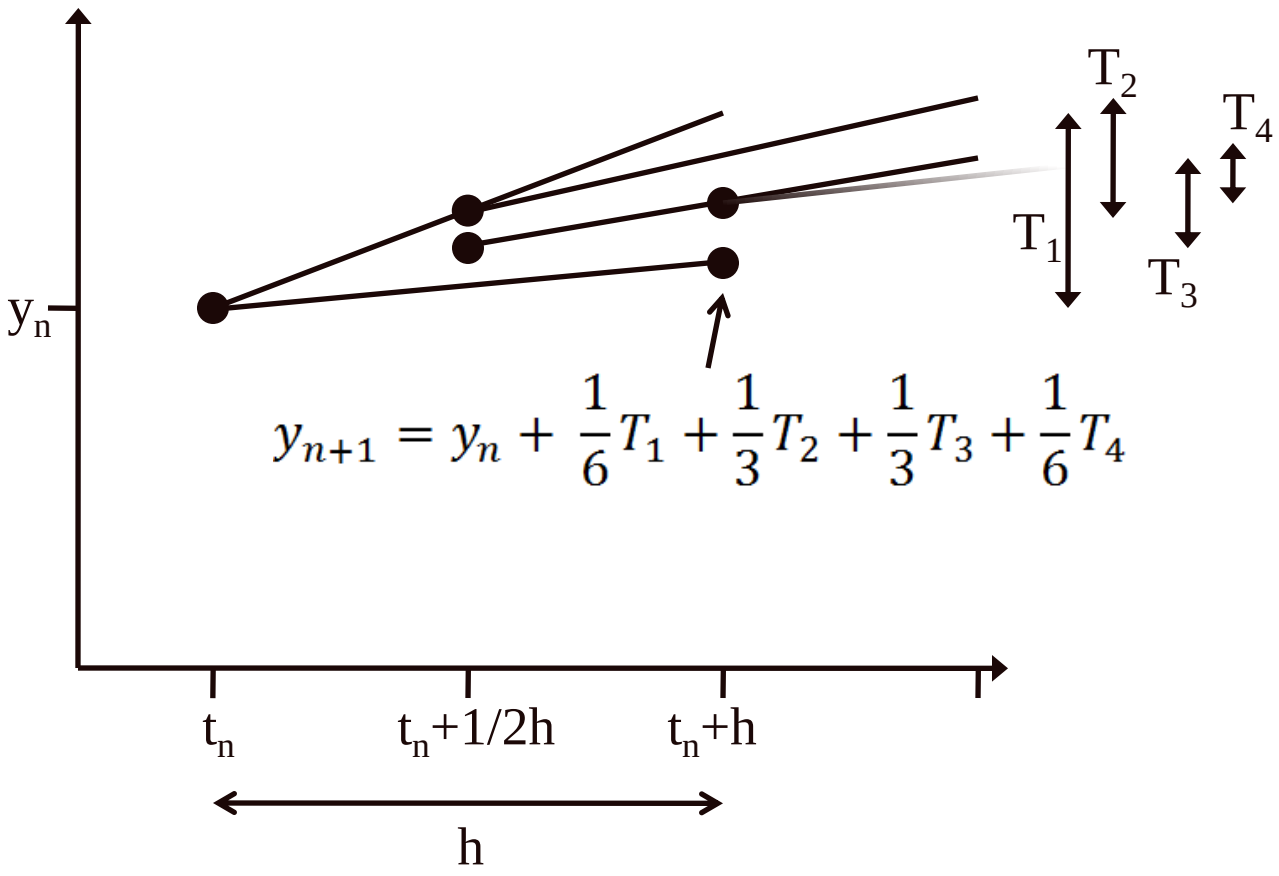
Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\frac{dy}{dt} = f(t, y)$$



Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4^{th} order Runge-Kutta scheme to advance in time:

Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\int_\Omega qT_1 = \left(\int_\Omega \nabla q \cdot \mathbf{u^n} T - qT\nabla \cdot \boldsymbol{u^n}\, dX \right) T^n$$

$$\int_\Omega qT_2 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_1}{2} \right)$$

$$\int_\Omega qT_3 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_2}{2} \right)$$

$$\int_\Omega qT_4 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+1}} T - qT\nabla \cdot \boldsymbol{u^{n+1}}\, dX \right) (T^n + T_3)$$

Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\int_\Omega qT_1 = \left(\int_\Omega \nabla q \cdot \mathbf{u^n} T - qT\nabla \cdot \boldsymbol{u^n}\, dX \right) T^n$$

$$\int_\Omega qT_2 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_1}{2} \right)$$

$$\int_\Omega qT_3 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_2}{2} \right)$$

$$\int_\Omega qT_4 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+1}} T - qT\nabla \cdot \boldsymbol{u^{n+1}}\, dX \right) (T^n + T_3)$$

$$\int_\Omega qT^{n+1}\, dX = \int_\Omega qT^n\, dX + \frac{1}{6}T_1 + \frac{1}{3}T_2 + \frac{1}{3}T_3 + \frac{1}{6}T_4$$

Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\int_\Omega qT_1 = \left(\int_\Omega \nabla q \cdot \mathbf{u^n} T - qT\nabla \cdot \boldsymbol{u^n}\, dX \right) T^n$$

$$\int_\Omega qT_2 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_1}{2} \right)$$

$$\int_\Omega qT_3 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_2}{2} \right)$$

$$\int_\Omega qT_4 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+1}} T - qT\nabla \cdot \boldsymbol{u^{n+1}}\, dX \right) (T^n + T_3)$$

$$\int_\Omega qT^{n+1}\, dX = \int_\Omega qT^n\, dX + \frac{1}{6}T_1 + \frac{1}{3}T_2 + \frac{1}{3}T_3 + \frac{1}{6}T_4$$

Failure to synthesise:
Only integrals may be added to integrals
In domain terms:
Operands in different spaces (*test* & *trial*)

Imperial College London

"Type Checking" Advection-Diffusion

$$\frac{\partial T}{\partial t} + u\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$

Discretisation using a 4th order Runge-Kutta scheme to advance in time:

$$\int_\Omega qT_1 = \left(\int_\Omega \nabla q \cdot \mathbf{u^n} T - qT\nabla \cdot \boldsymbol{u^n}\, dX \right) T^n$$

$$\int_\Omega qT_2 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_1}{2} \right)$$

$$\int_\Omega qT_3 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+\frac{1}{2}}} T - qT\nabla \cdot \boldsymbol{u^{n+\frac{1}{2}}}\, dX \right) \left(T^n + \frac{T_2}{2} \right)$$

$$\int_\Omega qT_4 = \left(\int_\Omega \nabla q \cdot \mathbf{u^{n+1}} T - qT\nabla \cdot \boldsymbol{u^{n+1}}\, dX \right) (T^n + T_3)$$

$$\int_\Omega qT^{n+1}\, dX = \int_\Omega qT^n\, dX + \frac{1}{6}T_1 + \frac{1}{3}T_2 + \frac{1}{3}T_3 + \frac{1}{6}T_4$$

Failure to synthesise:
Only integrals may be added to integrals
In domain terms:
Operands in different spaces (*test* & *trial*)

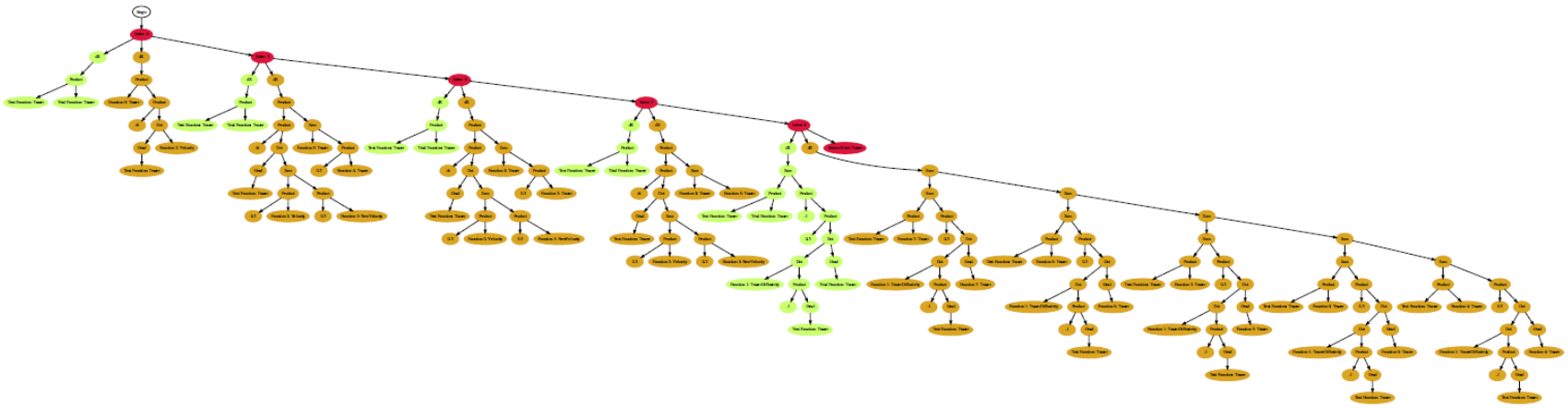
The correct way:

$$T^{n+1} = T^n + \frac{1}{6}T_1 + \frac{1}{3}T_2 + \frac{1}{3}T_3 + \frac{1}{6}T_4$$

Imperial College London

Ongoing work

- Completion of the CUDA code generator
- Advection-diffusion IR:



- Exploitation of optimisation opportunities
- Targeting OpenCL, x86/SSE/multicore
- Shallow Water Equations code generation

Conclusions

- UFL is the right abstraction for generating multiple-targeted FE solvers
- The mathematician's view:
  - You get to write down the maths
  - Reduces code length by an order of magnitude
  - Catches nasty bugs that scientists really make
- The compiler engineer's view:
  - High-level language
  - High-performance opportunities
- Commitment to implementation in Fluidity

Imperial College London

References

The FEniCS project:
A. Logg, *Automating the Finite Element Method*, Arch. Comput. Metods Eng. 14 (2) (2007) 93-138

Addto vs. LMA on CPU implementations:
P. E. J. Vos, S. J. Sherwin, R. M. Kirby, *From h to p efficiently: implementing finite and spectral/hp element discretisations to achieve optimal performance at low and high order approximations*, Submitted to Journal of Computational Physics. (October 2009). http://www2.imperial.ac.uk/ssherw/spectralhp/papers/JCP-VoShKi-09.pdf

The finite element method:
G. E. M. Karniadakis, S. J. Sherwin, *Spectral/hp Element Methods for CFD*, Oxford University Press, 1999

Implementation of the test problem:
G. Markall, *Generatively Programming Galerkin Projections on General Purpose Graphics Processing Units*, Master's thesis, Imperial College London, http://www.doc.ic.ac.uk/~grm08/graham_markall_msc_report.pdf (2009).