Generating high-performance multiplatform finite element solvers using the Manycore Form Compiler and OP2

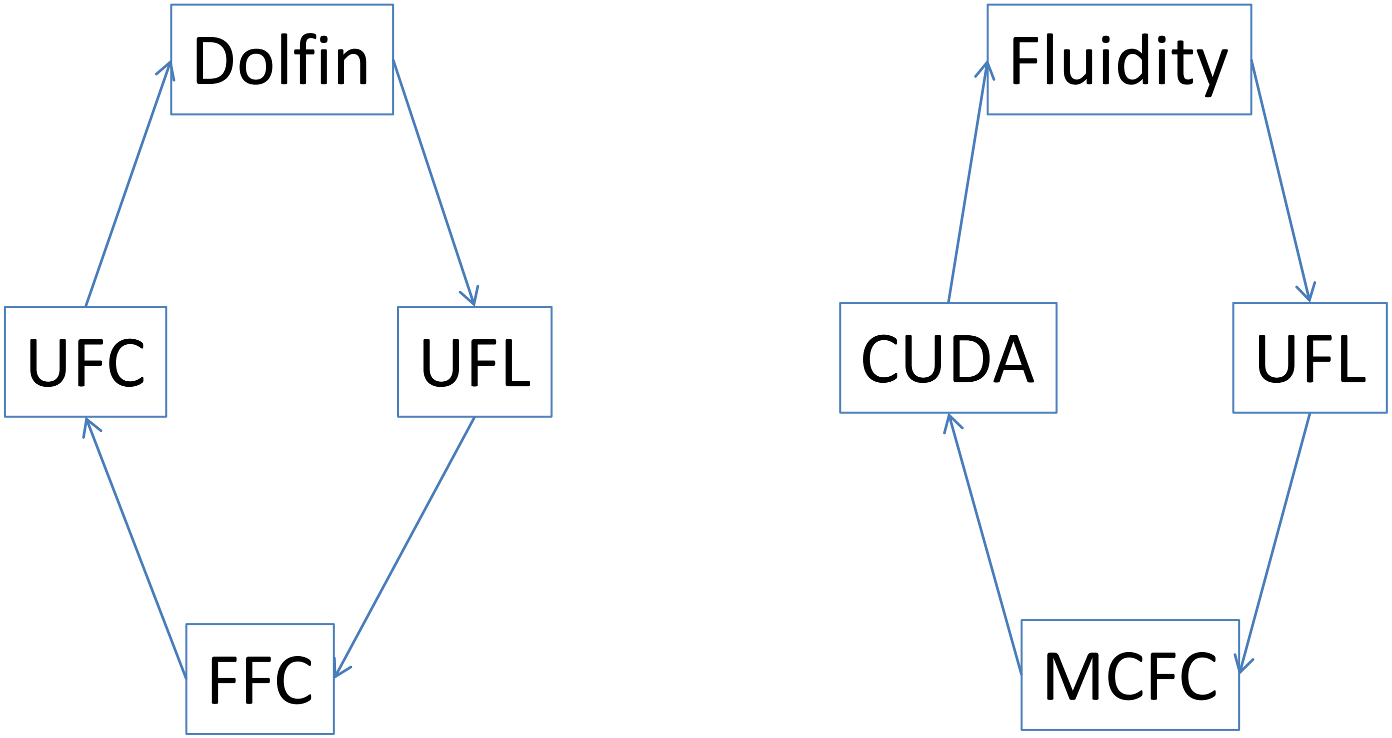
Graham R. Markall, Florian Rathgeber, David A. Ham, Paul H. J. Kelly, Carlo Bertolli, Adam Betts
Imperial College London
Mike B. Giles, Gihan R. Mudalige
University of Oxford
Istvan Z. Reguly
Pazmany Peter Catholic University, Hungary
Lawrence Mitchell
University of Edinburgh

- How do we get performance portability for the finite element method?

- Using a form compiler with pluggable backend support
  - One backend: CUDA – NVidia GPUs

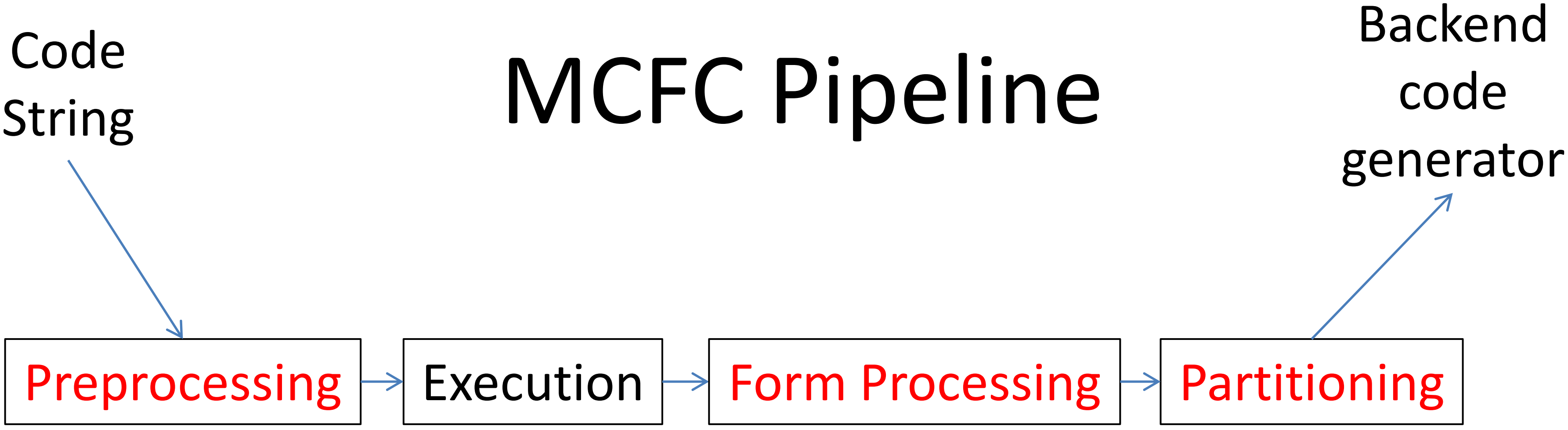
- Long term plan:
  - Target an *intermediate representation*

Manycore Form Compiler



- Compile-time code generation
  - Plans to move to runtime code generation
- Generates assembly and marshalling code
- Designed to support isoparametric elements

MCFC Pipeline



- Preprocessing: insert Jacobian and transformed gradient operators into forms
- Execution: Run in python interpreter, retrieve `Form` objects from namespace
- Form processing: `compute_form_data()`
- Partitioning: helps loop-nest generation

Preprocessing

- Handles coordinate transformation as part of the form using UFL primitives

```
x = state.vector_fields['Coordinate']
J = Jacobian(x)
invJ = Inverse(J)
detJ = Determinant(J)
```

- Multiply each form by J
- Overloaded derivative operators, e.g.:

```
def grad(u):
    return ufl.dot(invJ, ufl.grad(u))
```

- Code generation gives no special treatment to the Jacobian, its determinant or inverse

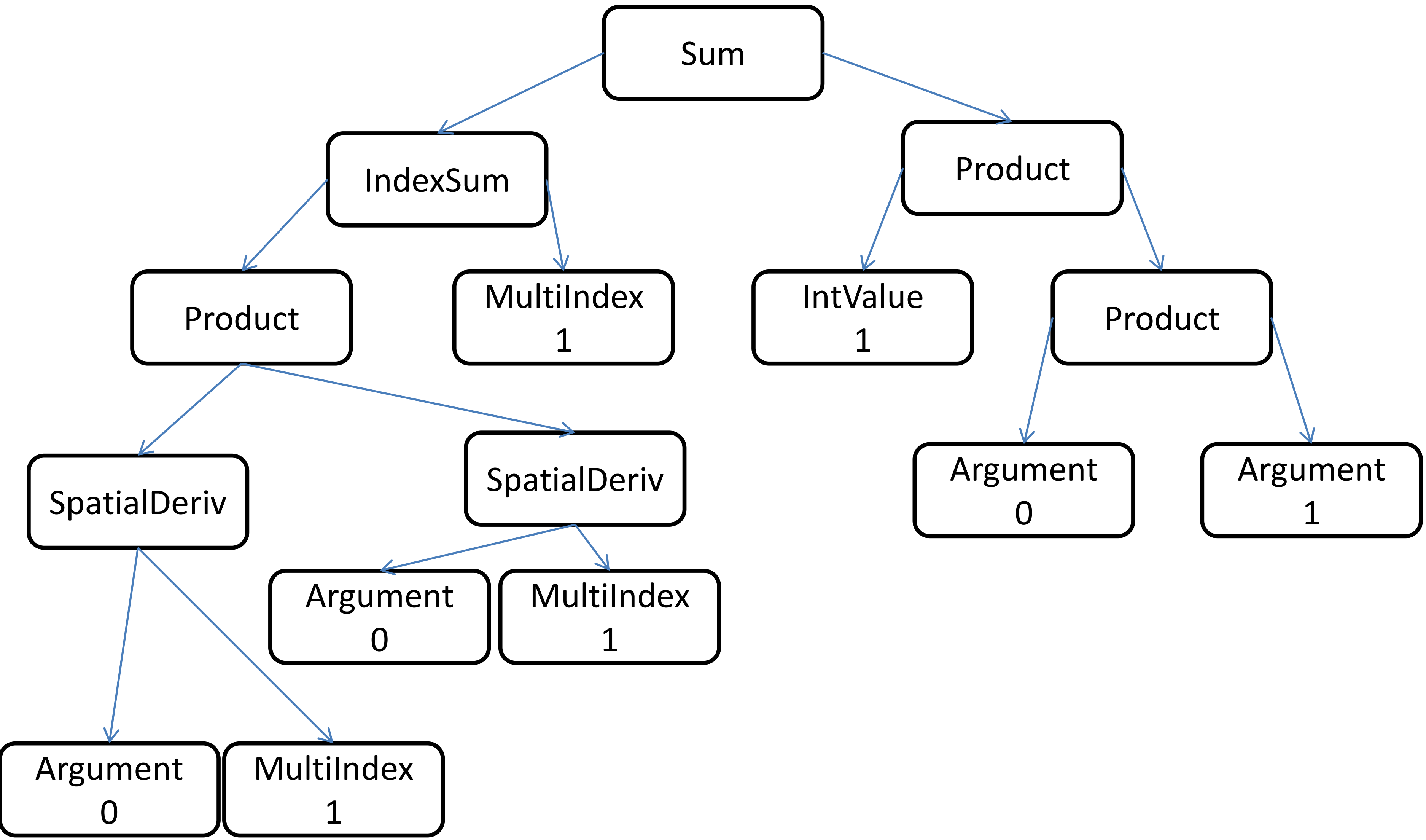
Loop nest generation

- Loops in typical assembly kernel:

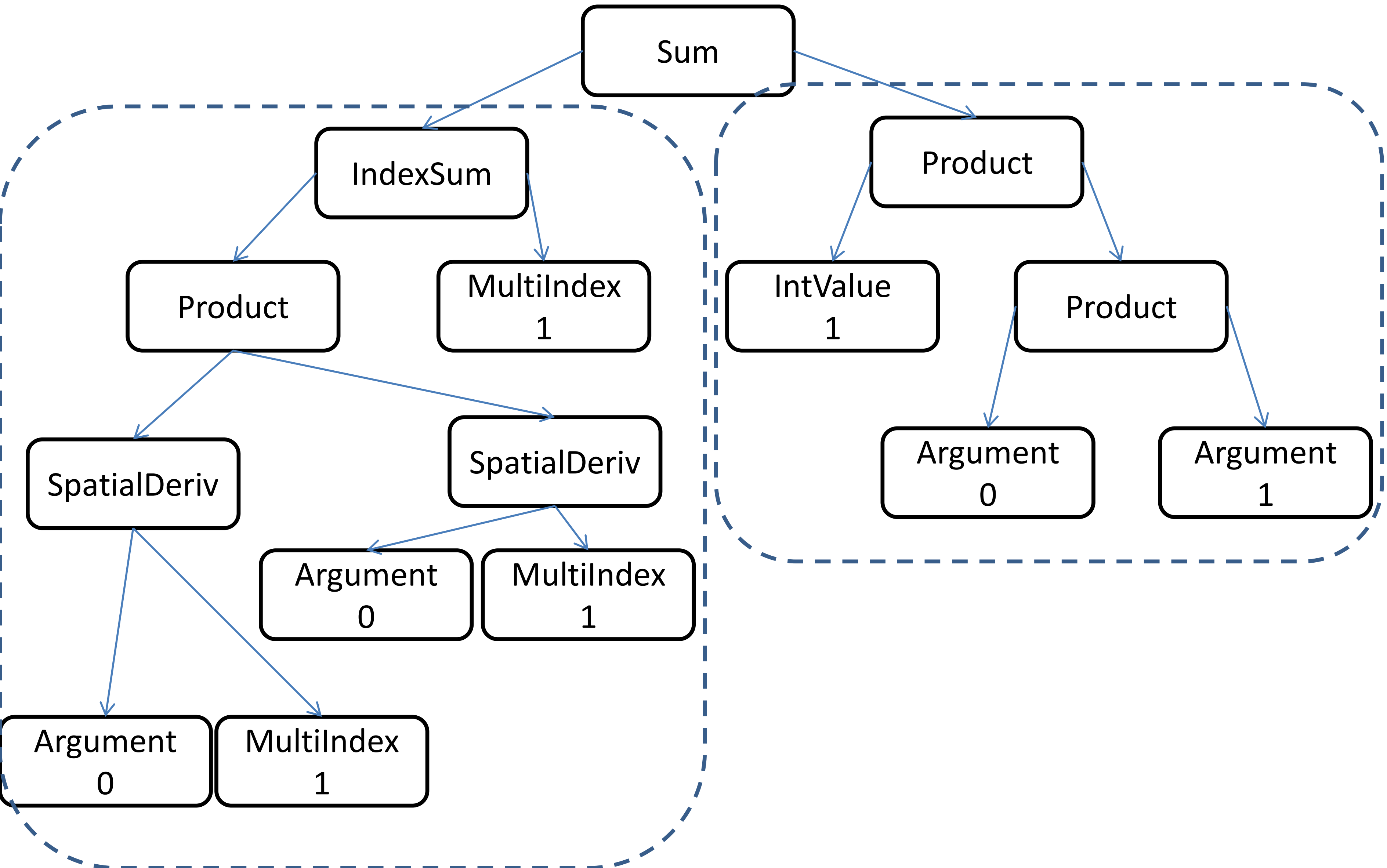
```
For (int i=0; i<3; ++i)
  For (int j=0; j<3; ++j)
    for (int q=0; q<6; ++q)
      for (int d=0; d<2; ++d)
```

- Inference of loop structure from preprocessed form:
  - Basis functions: use rank of form
  - Quadrature loop: Quadrature degree known
  - Dimension loops:
    - Find all the IndexSum indices
    - Recursively descend through form graph identifying maximal sub-graphs that share sets of indices

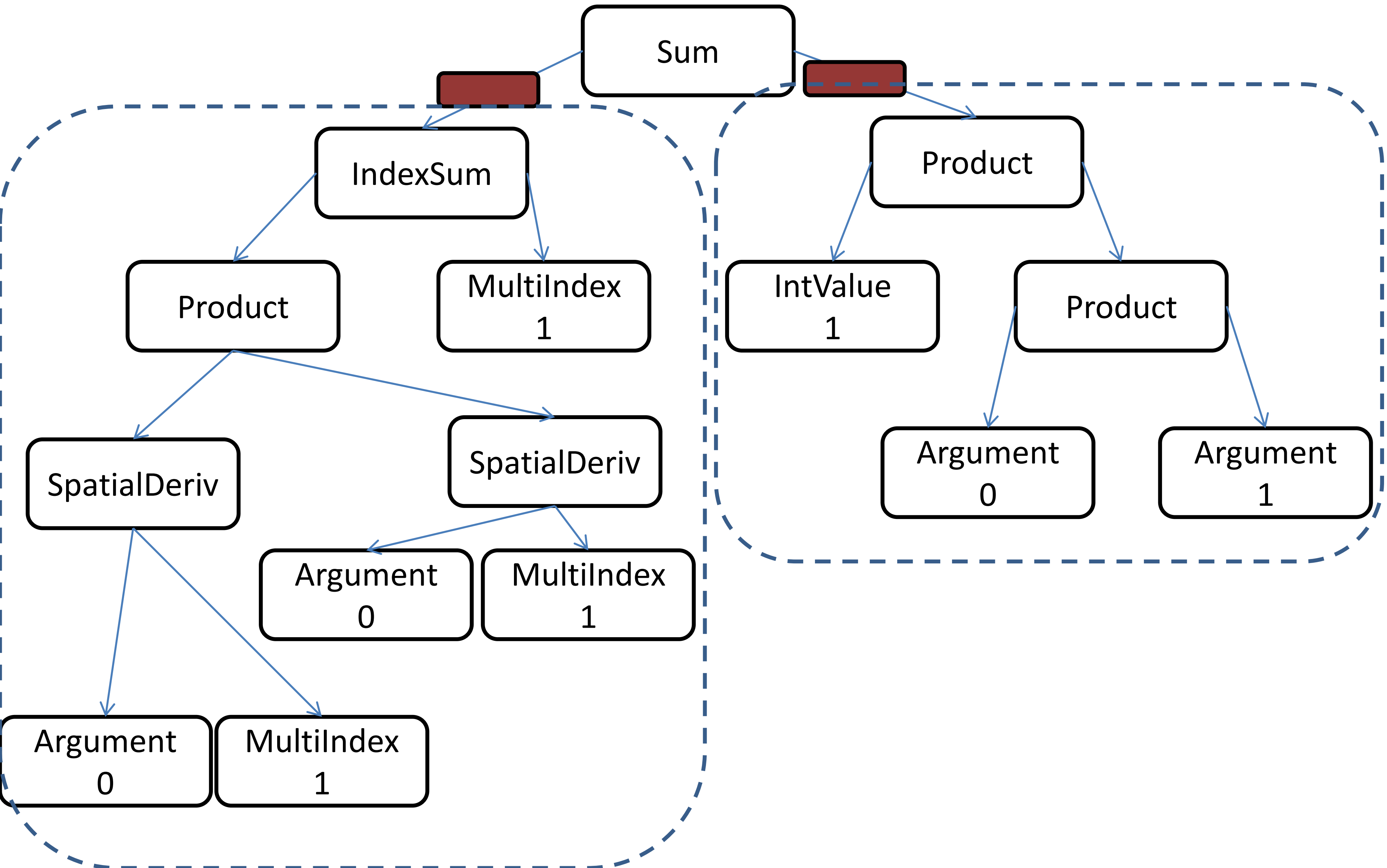
Partitioning example: $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, dX$



Partitioning example: $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, dX$



Partitioning example: $\int_{\Omega} \nabla v \cdot \nabla u + \lambda v u \, dX$

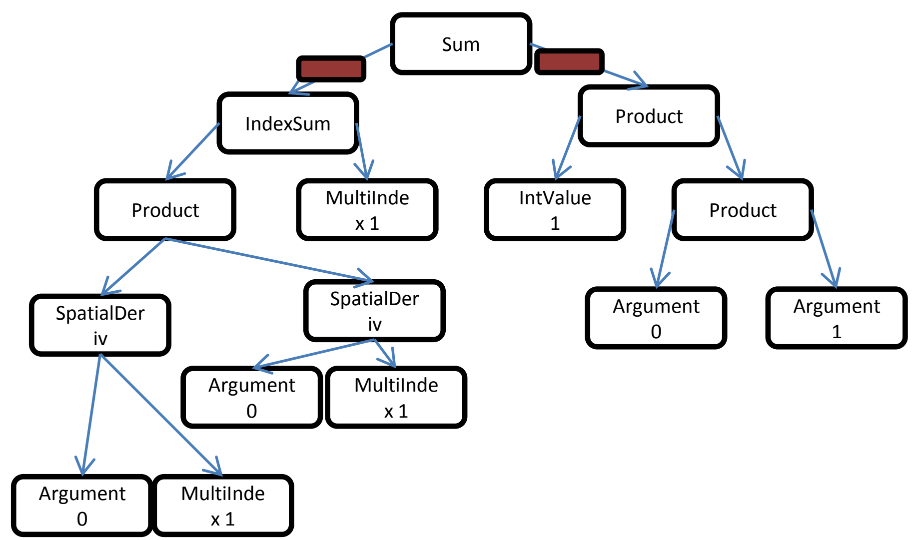


Partition code generation

- Once we know which loops to generate:
  - Generate an expression for each partition (*subexpression*)
  - Insert the subexpression into the loop nest depending on the indices it refers to
  - Traverse the topmost expression of the form, and generate an expression that combines subexpressions, and insert into loop nest

Code gen example: $\int_{\Omega} \nabla v \cdot \nabla u + \lambda v u \, dX$

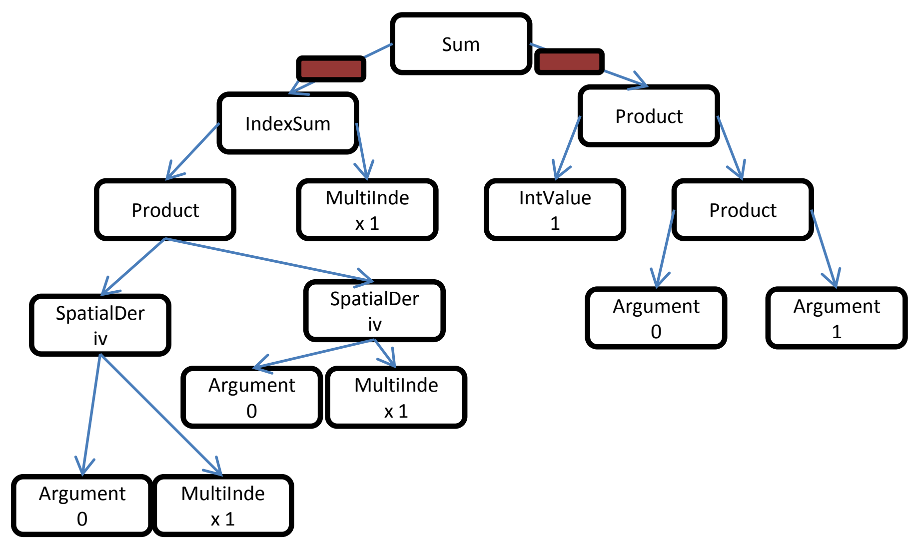
```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {

    for (int q=0; q<6; ++q) {




      for (int d=0; d<2; ++d) {

      }

    }
  }
}
```



Code gen example:

$$\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, dX$$

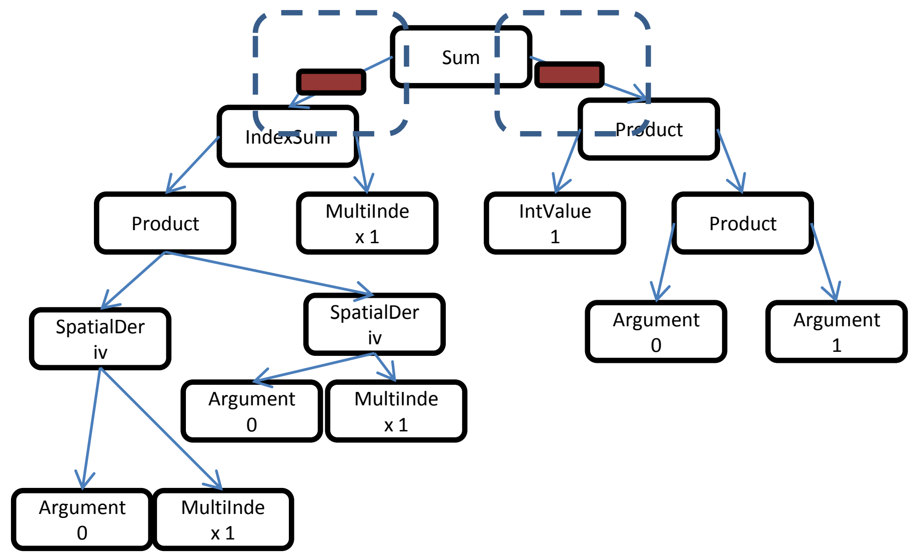
```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    LocalTensor[i,j] = 0.0;
    for (int q=0; q<6; ++q) {




      for (int d=0; d<2; ++d) {

      }

    }
  }
}
```



Code gen example:

$$\int_{\Omega} \nabla v \cdot \nabla u + \lambda v u \, dX$$

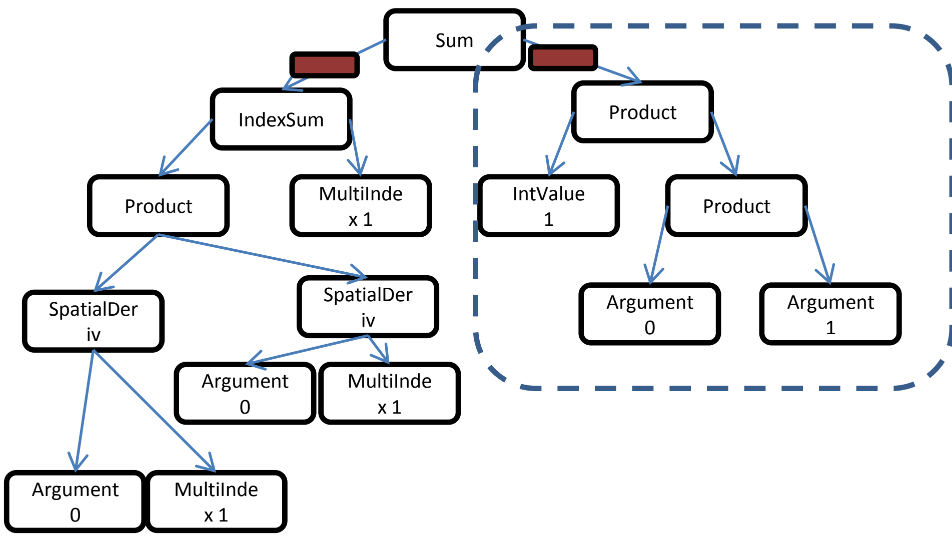
```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    LocalTensor[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr0 = 0.0
      SubExpr1 = 0.0

      for (int d=0; d<2; ++d) {

      }

    }
  }
}
```



Code gen example:

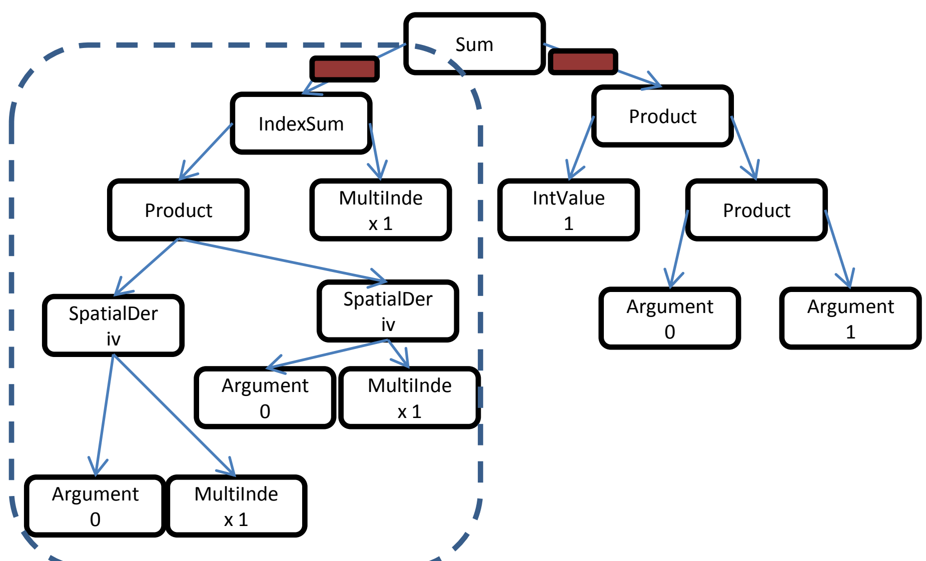
$$\int_{\Omega} \nabla v \cdot \nabla u + \lambda v u \, dX$$

```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    LocalTensor[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr0 = 0.0
      SubExpr1 = 0.0
      SubExpr0 += arg[i,q]*arg[j,q]
      for (int d=0; d<2; ++d) {

      }

    }
  }
}
```



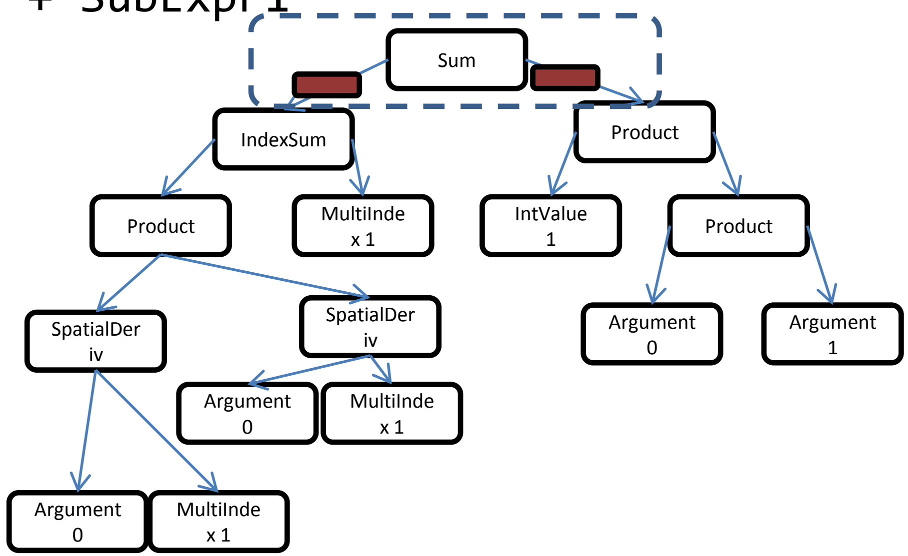
Code gen example: $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, dX$

```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    LocalTensor[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr0 = 0.0
      SubExpr1 = 0.0
      SubExpr0 += arg[i,q]*arg[j,q]
      for (int d=0; d<2; ++d) {
        SubExpr1 += d_arg[d,i,q]*d_arg[d,j,q]
      }

    }
  }
}
```



Code gen example: $\int_\Omega \nabla v \cdot \nabla u + \lambda v u \, dX$

```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j) {
    LocalTensor[i,j] = 0.0;
    for (int q=0; q<6; ++q) {
      SubExpr0 = 0.0
      SubExpr1 = 0.0
      SubExpr0 += arg[i,q]*arg[j,q]
      for (int d=0; d<2; ++d) {
        SubExpr1 += d_arg[d,i,q]*d_arg[d,j,q]
      }
      LocalTensor[i,j] += SubExpr0 + SubExpr1
    }
  }
}
```



Benchmarking MCFC and DOLFIN

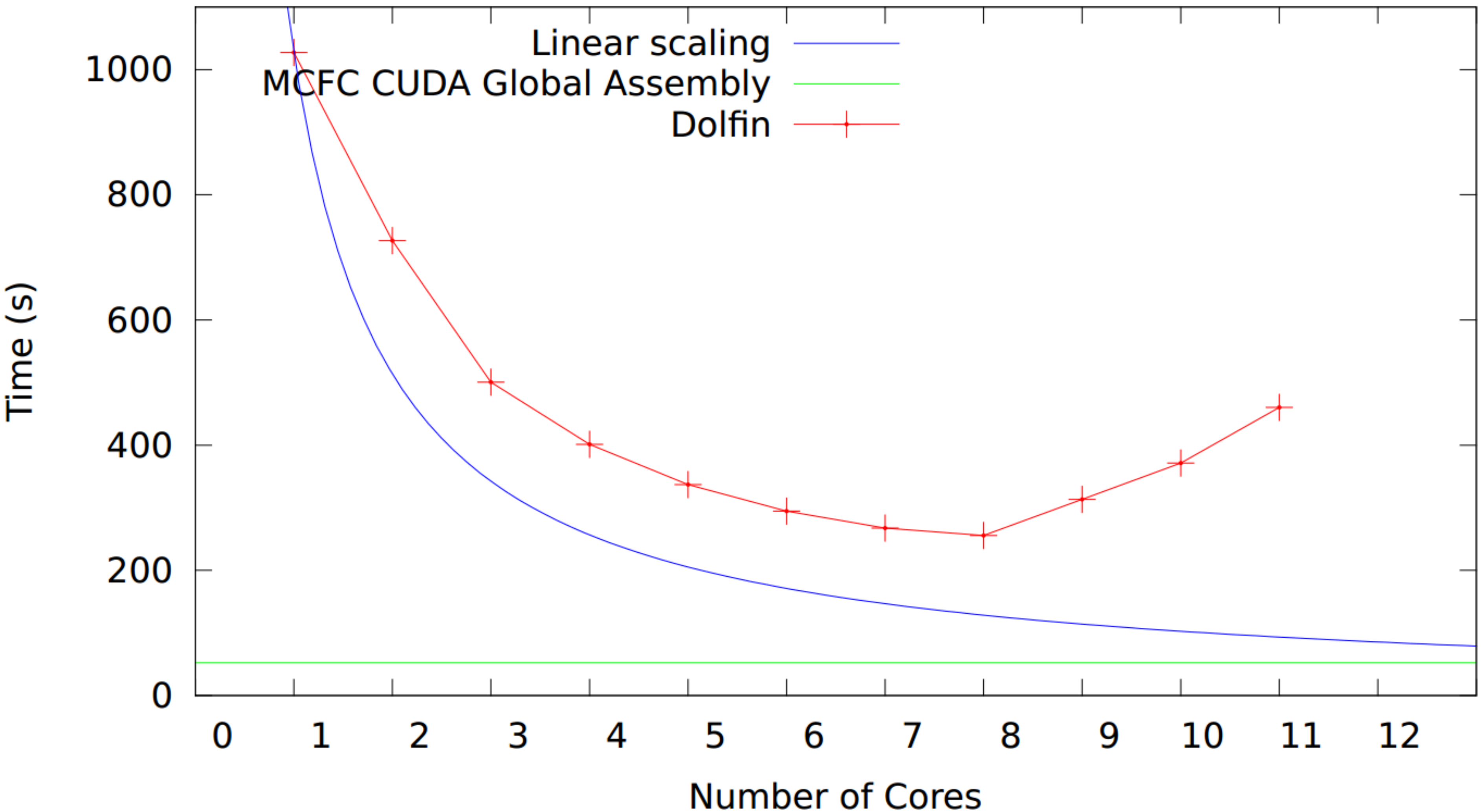
- Comparing and profiling assembly + solve of an advection-diffusion test case:

```
Coefficient(FiniteElement(“CG”, "triangle", 1))
p=TrialFunction(t)
q=TestFunction(t)

diffusivity = 0.1

M=p*q*dx

adv_rhs = (q*t+dt*dot(grad(q),u)*t)*dx
t_adv = solve(M, adv_rhs)
d=-dt*diffusivity*dot(grad(q),grad(p))*dx

A=M-0.5*d
diff_rhs=action(M+0.5*d,t_adv)
tnew=solve(A,diff_rhs)
```

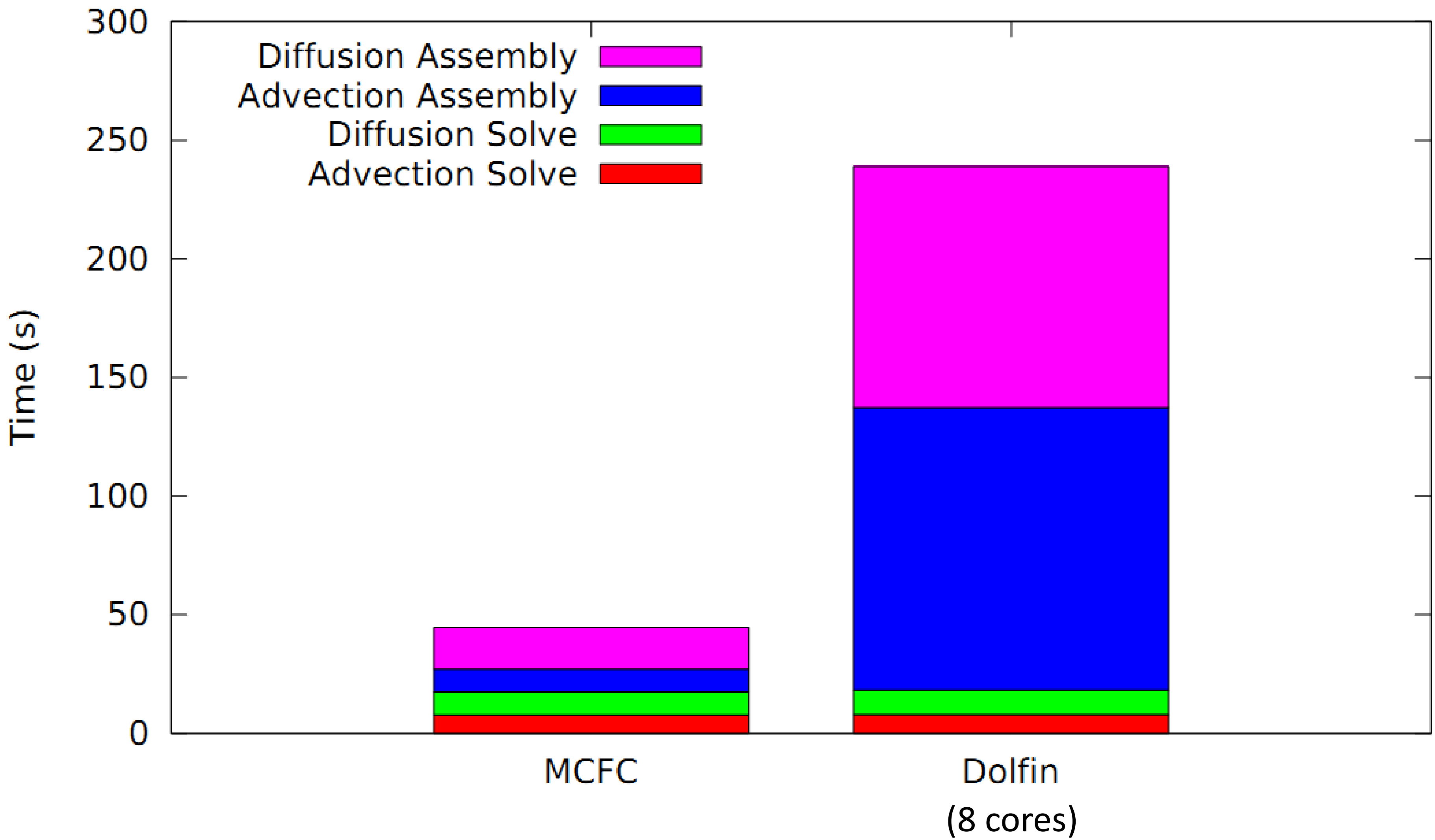
Experiment setup

- Term-split advection-diffusion equation
  - Advection: Euler timestepping
  - Diffusion: Implicit theta scheme
- Solver: CG with Jacobi preconditioning
  - Dolfin: PETSc
  - MCFC: From (Markall, 2009)
- CPU: 2 x 6 core Intel Xeon E5650 Westmere (HT off), 48GB RAM
- GPU Nvidia GTX480
- Mesh: 344128 unstructured elements, square domain. Run for 640 timesteps.
- Dolfin setup: Tensor representation, CPP opts on, form compiler opts off, MPI parallel

Adv-diff runtime



Breakdown of solver runtime



Dolfin profile

% Exec.	Function
15.8549	pair<boost::unordered_detail::hash_iterator_base<allocator<unsigned ... >::emplace()
11.9482	MatSetValues_MPIAIJ()
10.2417	malloc_consolidate
7.48235	_int_malloc
6.90363	dolfin::SparsityPattern::~SparsityPattern()
2.60801	dolfin::UFC::update()
2.48799	MatMult_SeqAIJ()
2.48758	ffc_form_d2c601cd1b0e28542a53997b6972359545bb30cc_cell_integral_0_0::tabulate_tensor()
2.3168	/usr/lib/openmpi/lib/libopen-pal.so.0.0.0
2.22407	boost::unordered_detail::hash_table<boost::unordered_detail::set<boost::hash<... >::rehash_impl()
1.9389	dolfin::MeshEntity::entities()
1.89775	_int_free
1.83794	free
1.71037	malloc
1.5123	/usr/lib/openmpi/lib/openmpi/mca_btl_sm.so
1.47677	/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.15
1.47279	poll
1.42863	ffc_form_958612b38a9044a3a64374d9d4be0681810fdbd8_cell_integral_0_0::tabulate_tensor()
1.18282	dolfin::SparsityPattern::insert()
1.13536	ffc_form_ba88085bc231bf16ec1c084f12b9c723279414f1_cell_integral_0_0::tabulate_tensor()
1.08694	ffc_form_23b22f19865ca4de78804edcf2815d350d5a55a3_cell_integral_0_0::tabulate_tensor()
0.983646	dolfin::GenericFunction::evaluate()
0.95484	dolfin::Function::restrict()
0.869109	VecSetValues_MPI()

MCFC CUDA Profile

% Exec.	Kernel
28.7	Matrix addto
14.9	Diffusion matrix local assembly
7.1	Vector addto
4.1	Diffusion RHS
2.1	Advection RHS
0.5	Mass matrix local assembly
42.6	Solver kernels

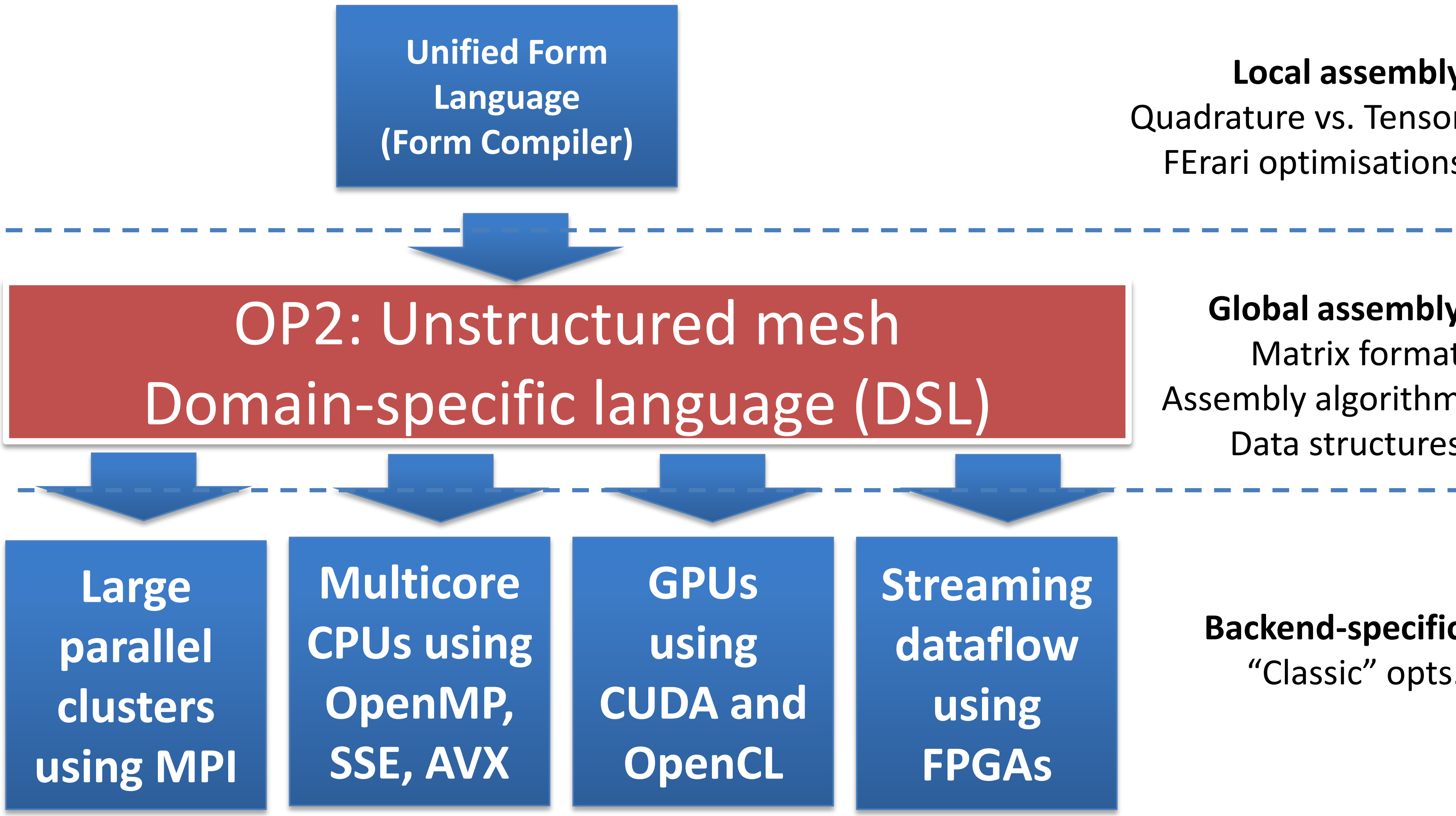
Thoughts

- Targeting the hardware directly allows for efficient implementations to be generated
- The MCFC CUDA backend embodies form-specific and hardware specific knowledge

- We need to target a performance portable *intermediate representation*

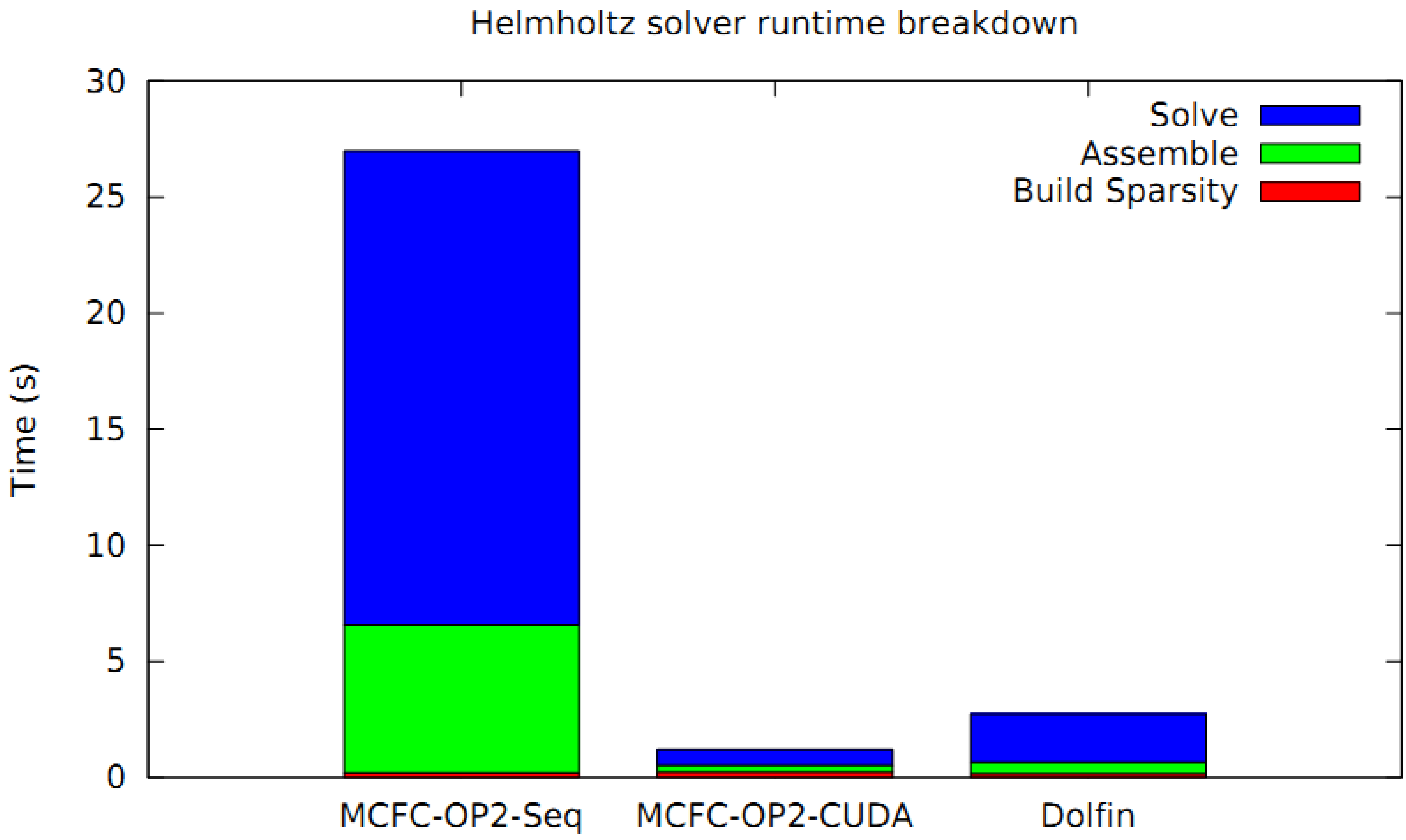
Layers manage complexity. Each layer of the IR:

- New optimisations introduced that are not possible in the higher layers
- With less complexity than the lower layers



Why OP2 for MCFC?

- Isolates a *kernel* that performs an operation for *every* mesh component – (Local Assembly)
- The job of OP2 is to control all code necessary to apply the kernel, fast
- Pushing all the OpenMP, MPI, OpenCL, CUDA, AVX issues into the OP2 compiler.
- Abstracts away the matrix representation so OP2 controls whether (and how/when) the matrix is assembled.



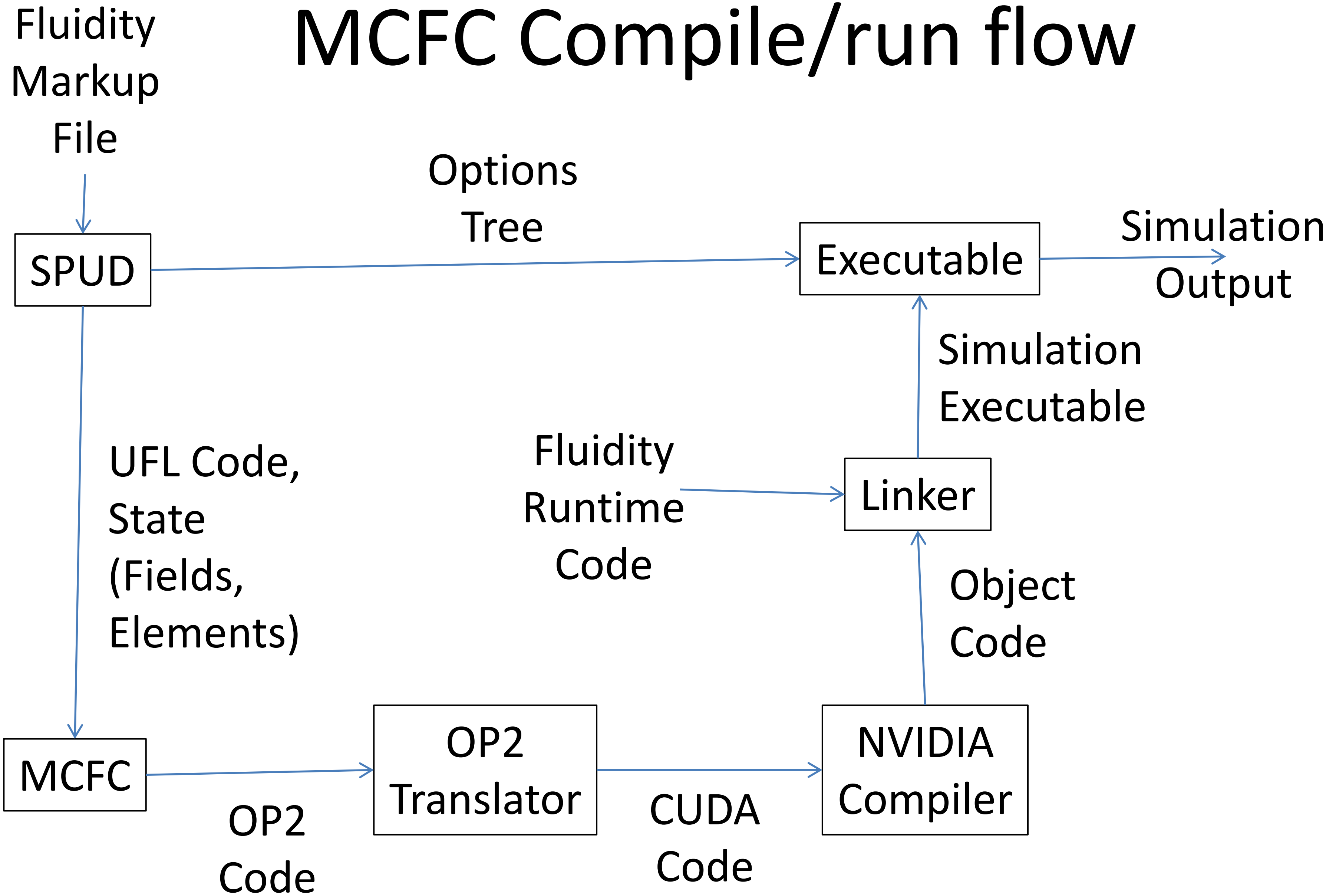
Helmholtz solver runtime breakdown
Time (s)
30
25
20
15
10
5
0
Solve
Assemble
Build Sparsity
MCFC-OP2-Seq
MCFC-OP2-CUDA
Dolfin

Summary

- High performance implementations are obtained by flattening out abstractions
- Flattening abstractions increases complexity – we need to combat this with a new, appropriate abstraction
- This greatly reduces the implementation space for the form compiler to work with
- Whilst still allowing *performance portability*
- MCFC OP2 implementation: ongoing

Spare slides

MCFC Compile/run flow



OP2 Matrix support

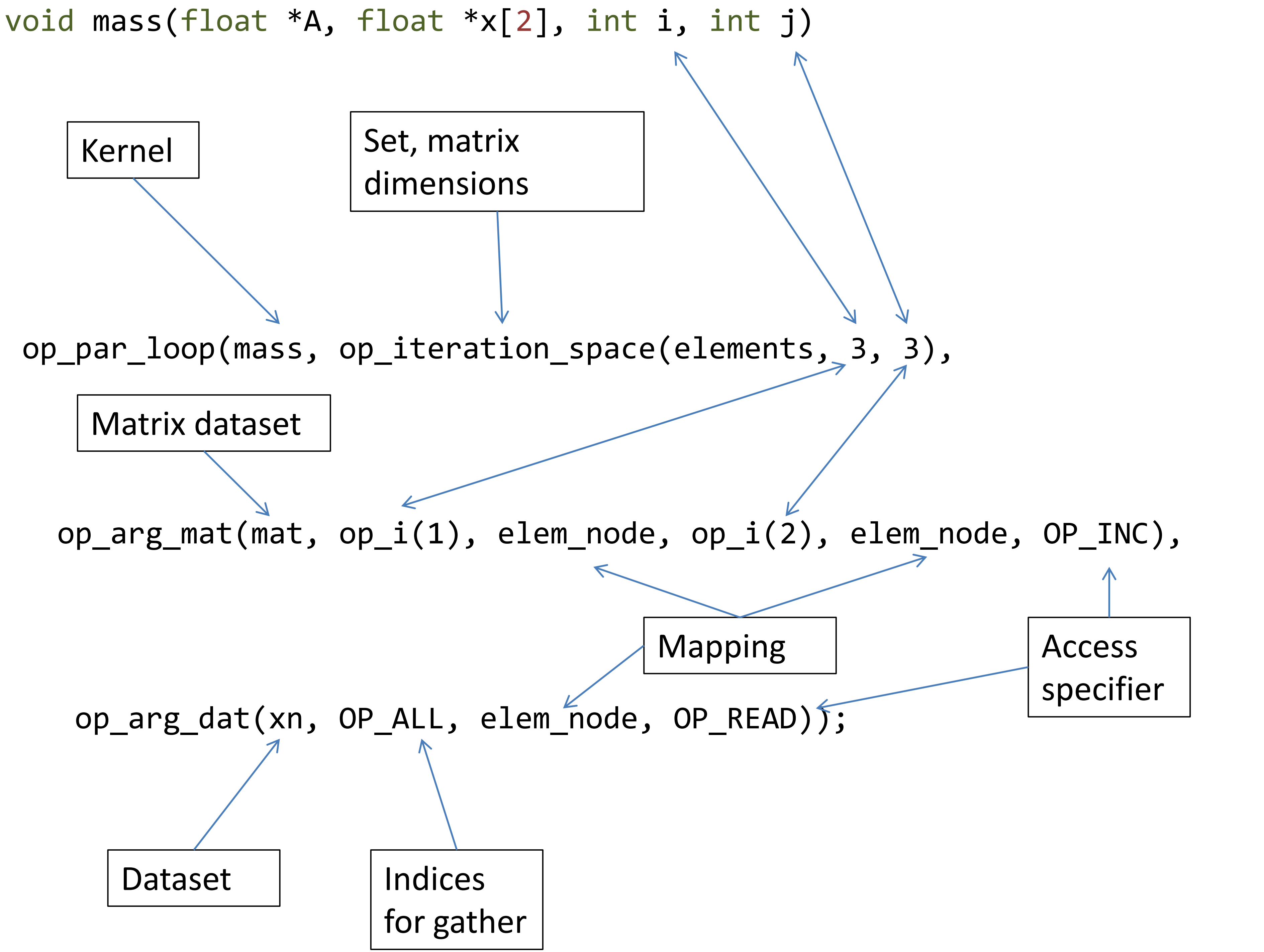
- *Matrix support* follows from Iteration Spaces:
  - What is the mapping between threads and elements? Example, on GPUs:
  - For low-order, one thread per element
  - For higher-order, one thread block per element
- OP2 extends iteration spaces to the matrix indices
- OP2 abstarcts them completely from the user – theyre inherently temposrary data types
- Theres no concept of getting the matrix back from op2.

```
void mass(float *A, float *x[2], int i, int j)
{
  int q;
  float J[2][2];
  float detJ;

  const float w[3]= {0.166667, 0.166667, 0.166667};
  const float CG1[3][3] = {{0.666667, 0.166667, 0.166667},
                           {0.166667, 0.666667, 0.166667},
                           {0.166667, 0.166667, 0.666667}};

  J[0][0] = x[1][0] - x[0][0];
  J[0][1] = x[2][0] - x[0][0];
  J[1][0] = x[1][1] - x[0][1];
  J[1][1] = x[2][1] - x[0][1];

  detJ = J[0][0] * J[1][1] - J[0][1] * J[1][0];

  for ( q = 0; q < 3; q++ )
    *A += CG1[i][q] * CG1[j][q] * detJ * w[q];
```

Pointer to a single matrix element (*A)

Ptr to coords of current element (*x[2])

Iteration space variables (i, j)


void mass(float *A, float *x[2], int i, int j)
Kernel
Set, matrix dimensions
op_par_loop(mass, op_iteration_space(elements, 3, 3),
Matrix dataset
op_arg_mat(mat, op_i(1), elem_node, op_i(2), elem_node, OP_INC),
Mapping
Access specifier
op_arg_dat(xn, OP_ALL, elem_node, OP_READ));
Dataset
Indices for gather

The OP2 abstraction

- The mesh is represented in a general manner as a graph. Primitives:
  - Sets (e.g. cells, vertices, edges)
  - mappings (e.g. from cells to vertices)
  - datasets (e.g. coefficients)

- No mesh entity requires special treatment
- Cells, vertices, etc are entities of different arity

The OP2 abstraction

- Parallel loops specify:
  - *A kernel*
  - An *Iteration space*: A set
  - An *Access Descriptor*: Datasets to pass to the kernel, and the mappings through which they're accessed

- OP2 Runtime handles application of the kernel at each point in the iteration space, feeding the data specified in the access descriptor