Imperial College London

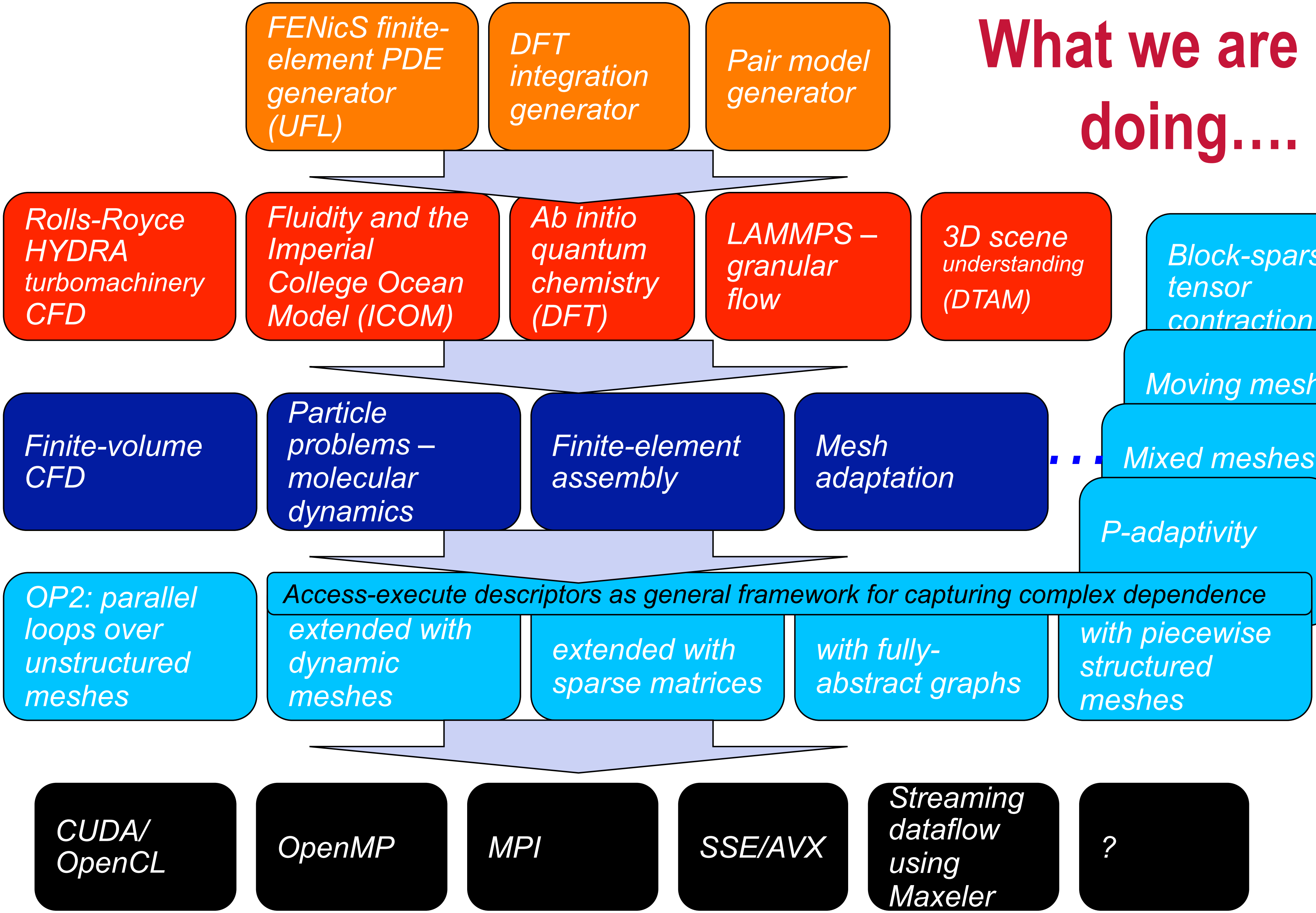
Generating programs works better than transforming them, if you get the abstraction right

Paul H J Kelly
Group Leader, Software Performance Optimisation
Co-Director, Centre for Computational Methods in Science and Engineering
Department of Computing
Imperial College London

Joint work with :

David Ham, Gerard Gorman, Florian Rathgeber (Imperial ESE/Grantham Inst for Climate Change Res)
Mike Giles, Gihan Mudalige (Mathematical Inst, Oxford)
Adam Betts, Carlo Bertolli, Nicolas Loriant, Graham Markall, George Rokos (Software Perf Opt Group, Imperial)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)

What we are doing….

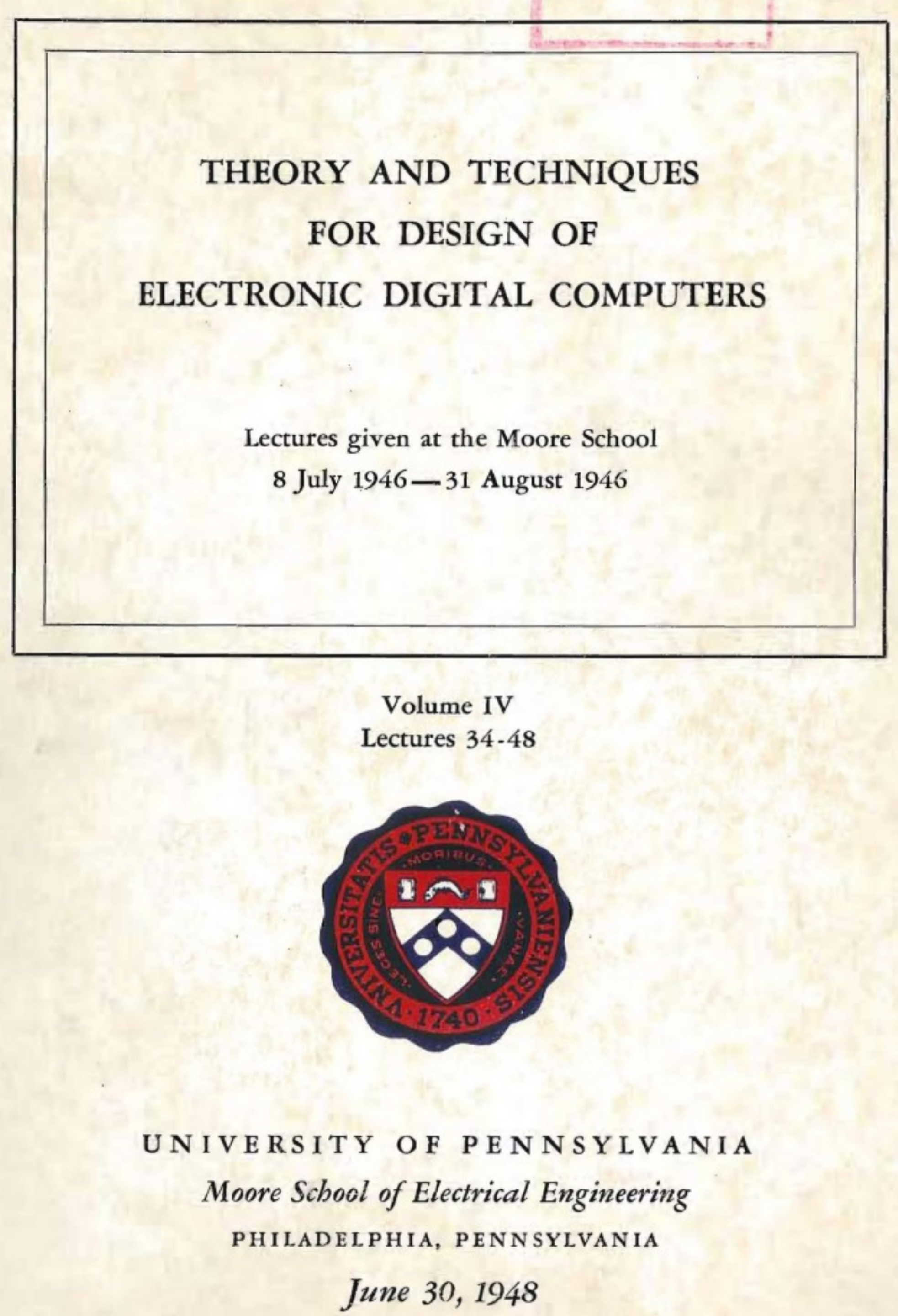


- Roadmap: applications drive DSLs, delivering performance portability

Imperial College London

The message

- Slogans
  - Generative, instead of transformative optimisation
  - Get the abstraction right, to isolate numerical methods from mapping to hardware
  - Build vertically, learn horizontally
  - Plenty of room at the top
  - The biggest opportunities are at the highest level
  - The value of generative and DSL techniques



THEORY AND TECHNIQUES FOR DESIGN OF ELECTRONIC DIGITAL COMPUTERS

Lectures given at the Moore School
8 July 1946—31 August 1946

Volume IV
Lectures 34-48

UNIVERSITY OF PENNSYLVANIA
Moore School of Electrical Engineering
PHILADELPHIA, PENNSYLVANIA
June 30, 1948

The Moore School Lectures

- *The first ever computer architecture conference*
- *July 8th to August 31st 1946, at the Moore School of Electrical Engineering, University of Pennsylvania*
- *A defining moment in the history of computing*

A PARALLEL CHANNEL COMPUTING MACHINE

Lecture by
J. P. Eckert, Jr.
Electronic Control Company

... Again I wish to reiterate the point that all the arguments for parallel operation are only valid provided one applies them to the steps which the built in or wired in programming of the machine operates. Any steps which are programmed by the operator, who sets up the machine, should be set up only in a serial fashion. It has been shown over and over again that any departure from this procedure results in a system which is much too complicated to use.

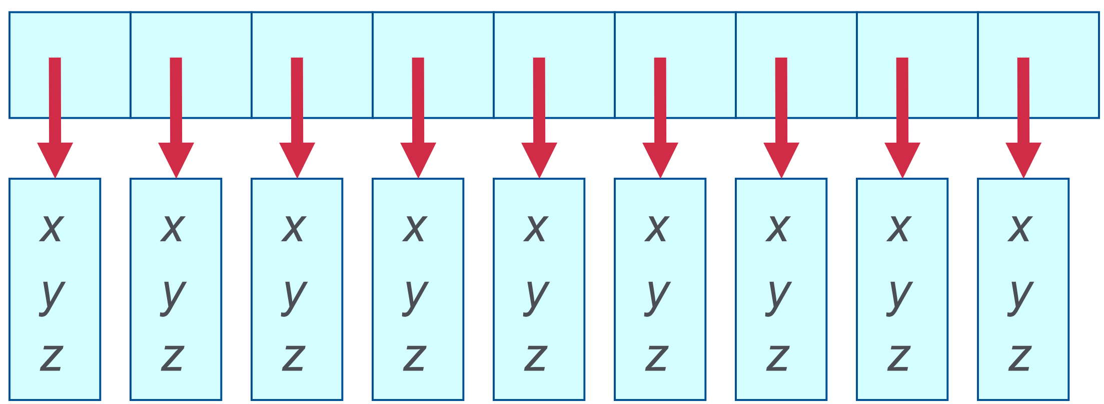
See also http://www.digital60.org/birth/themooreschool/lectures.html#l45

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- *Can the iterations of this loop be executed in parallel?*



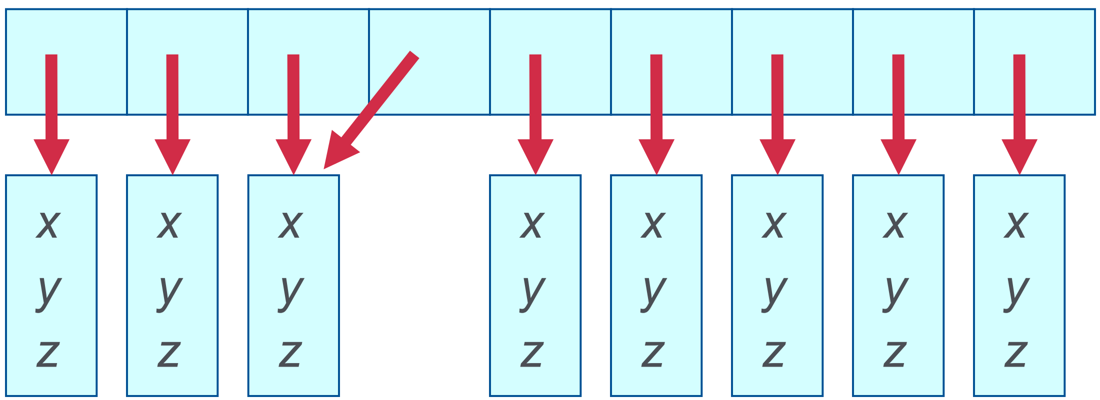
- *No problem: each iteration is independent*

Easy parallelism

Example:

```
for (i=0; i<N; ++i) {
  points[i]->x += 1;
}
```

- *Can the iterations of this loop be executed in parallel?*



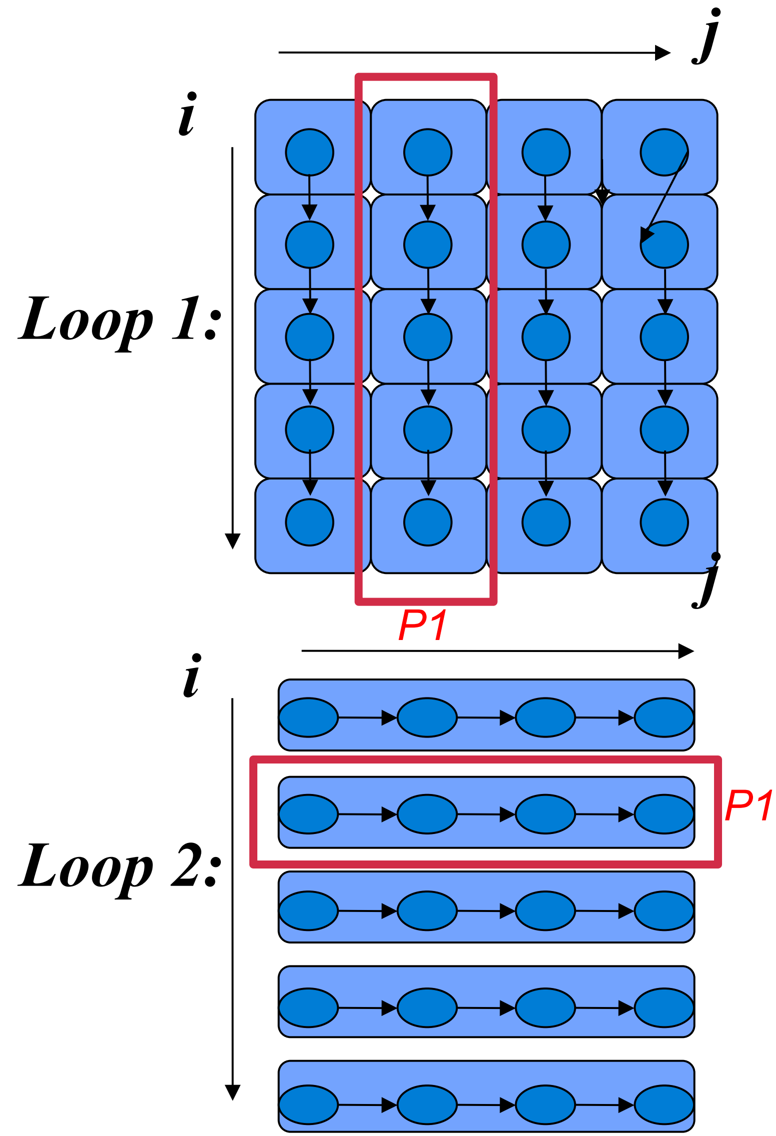
- *Oh no: not all the iterations are independent!*
  - *You want to re-use piece of code in different contexts*
  - *Whether it's parallel depends on context!*

Imperial College London

Another loss of abstraction…

- *Shared memory makes parallel programming much easier:*

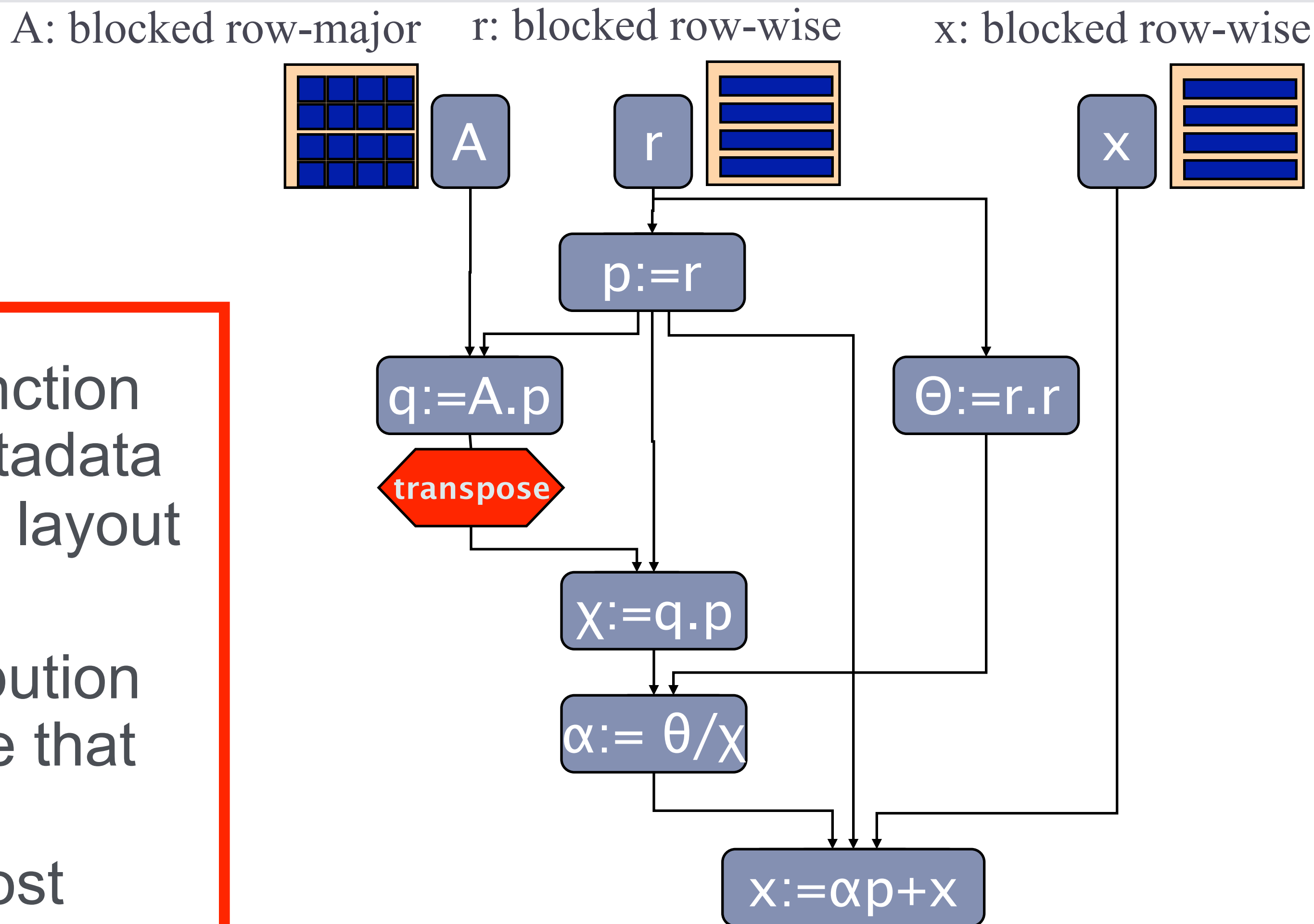
```
for(i=0; I<N; ++i)
   par_for(j=0; j<M; ++j)
      A[i,j] = (A[i-1,j] + A[i,j])*0.5;
par_for(i=0; I<N; ++i)
   for(j=0; j<M; ++j)
      A[i,j] = (A[i,j-1] + A[i,j])*0.5;
```



- *First loop operates on rows in parallel*
- *Second loop operates on columns in parallel*
- *With distributed memory we would have to program message passing to broadcast the columns in between*
- *With shared memory… no problem!*

Sarah Talbot and Paul Kelly, ***"Stable Performance for cc-NUMA using First Touch Page Placement and Reactive Proxies"****. HPCS'98*

Self-optimising linear algebra library

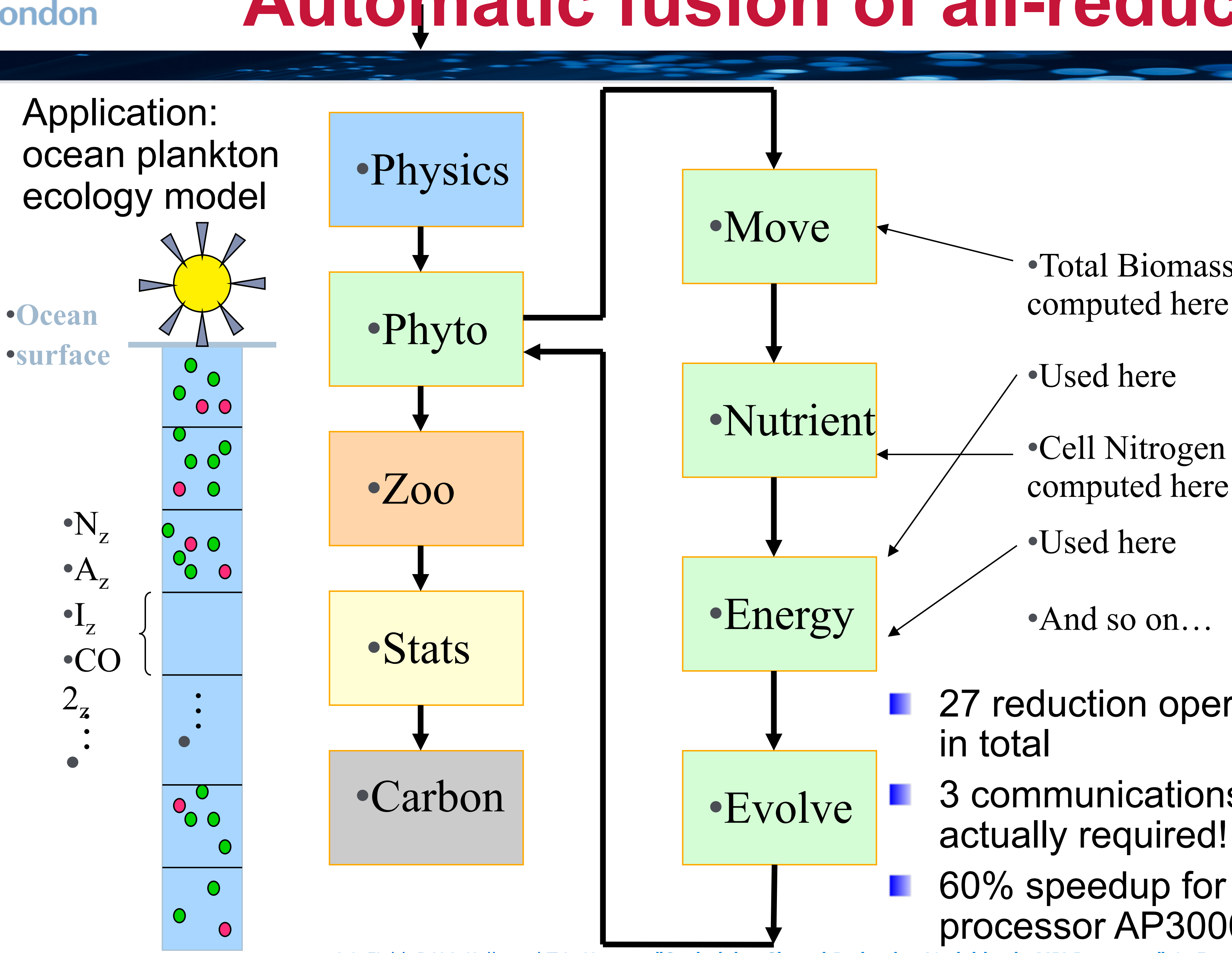


- Each library function comes with metadata describing data layout constraints
- Solve for distribution of each variable that minimises redistribution cost

(*Olav Beckmann and Paul H J Kelly, **"A Linear Algebra Formulation for Optimising Replication in Data Parallel Programs"**. LCPC'99*)

Automatic fusion of all-reduces

- Application: ocean plankton ecology model



- 27 reduction operations in total
- 3 communications actually required!
- 60% speedup for 32-processor AP3000

A.J. Field, P.H.J. Kelly and T.L. Hansen, **"Optimizing Shared Reduction Variables in MPI Programs"**. In Euro-Par 2002

Easy parallelism – tricky engineering

- *Parallelism breaks abstractions:*
  - *Whether code should run in parallel depends on context*
  - *How data and computation should be distributed across the machine depends on context*
- *"Best-effort", opportunistic parallelisation is almost useless:*
  - *Robust software must robustly, predictably, exploit large-scale parallelism*

How can we build robustly-efficient multicore software

While maintaining the abstractions that keep code clean, reusable and of long-term value?

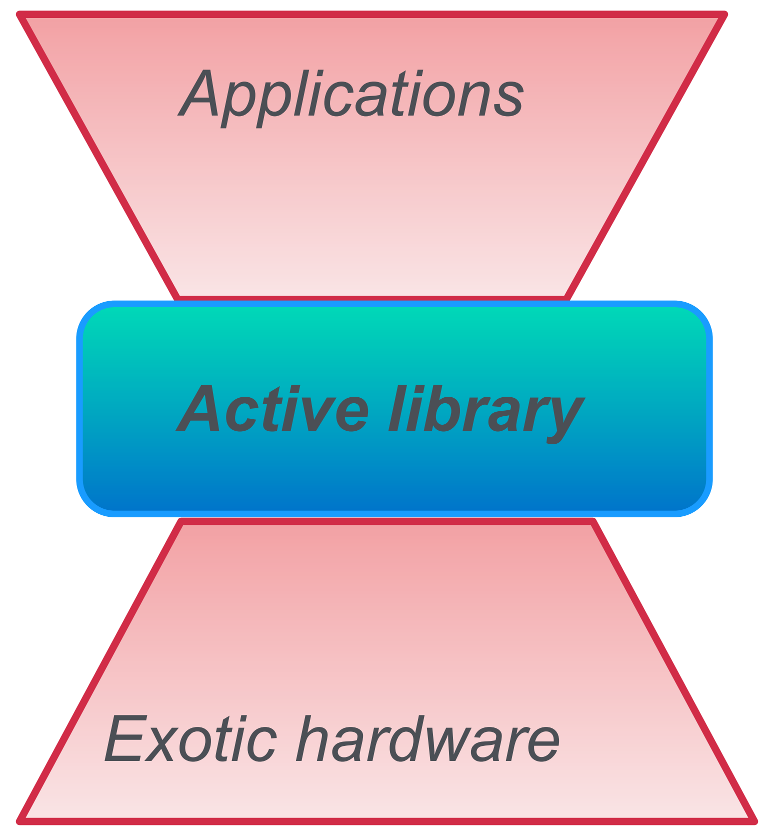
It's a software engineering problem

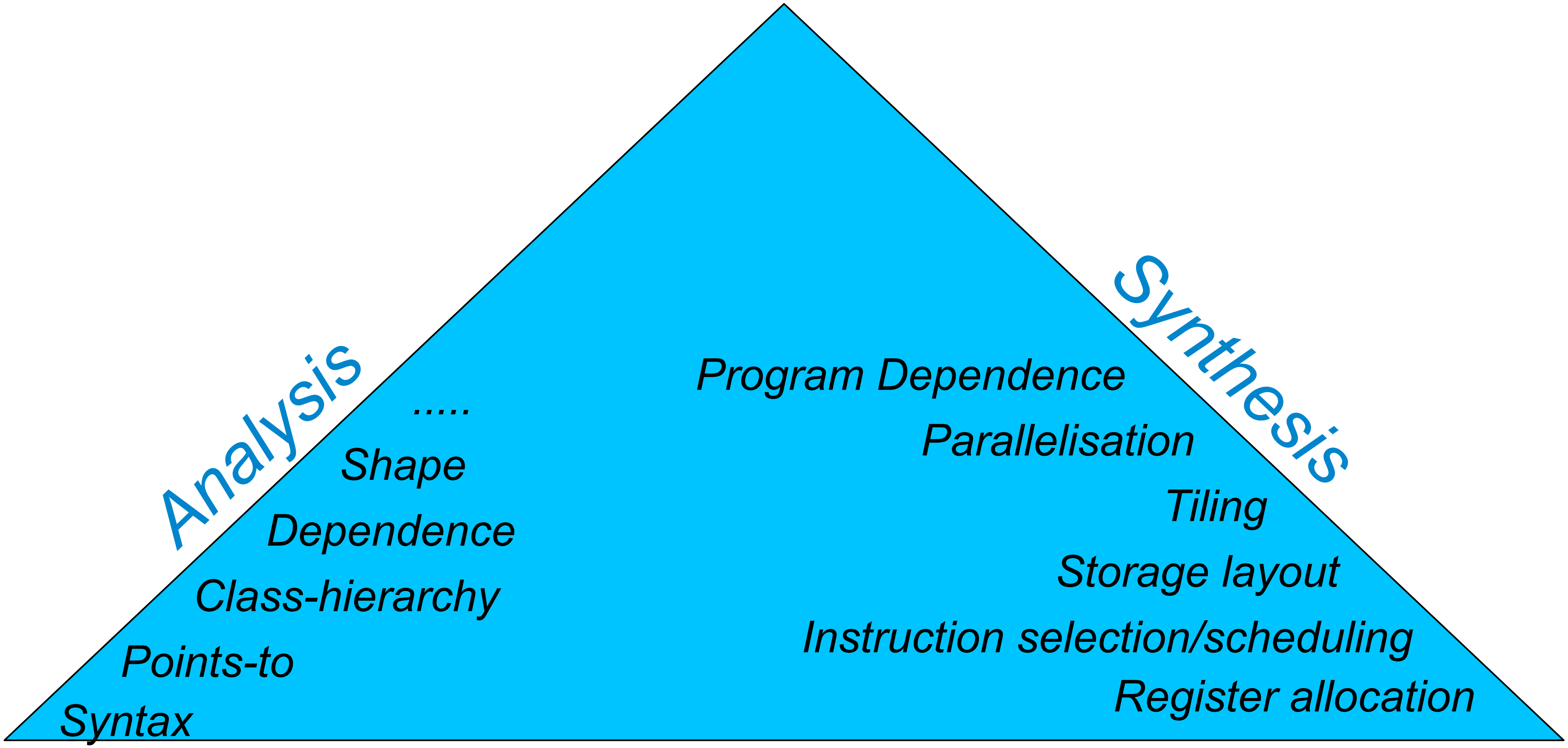
Active libraries and DSLs

Imperial College London

- Domain-specific languages...
- Embedded DSLs
- Active libraries
  - Libraries that come with a mechanism to deliver library-specific optimisations

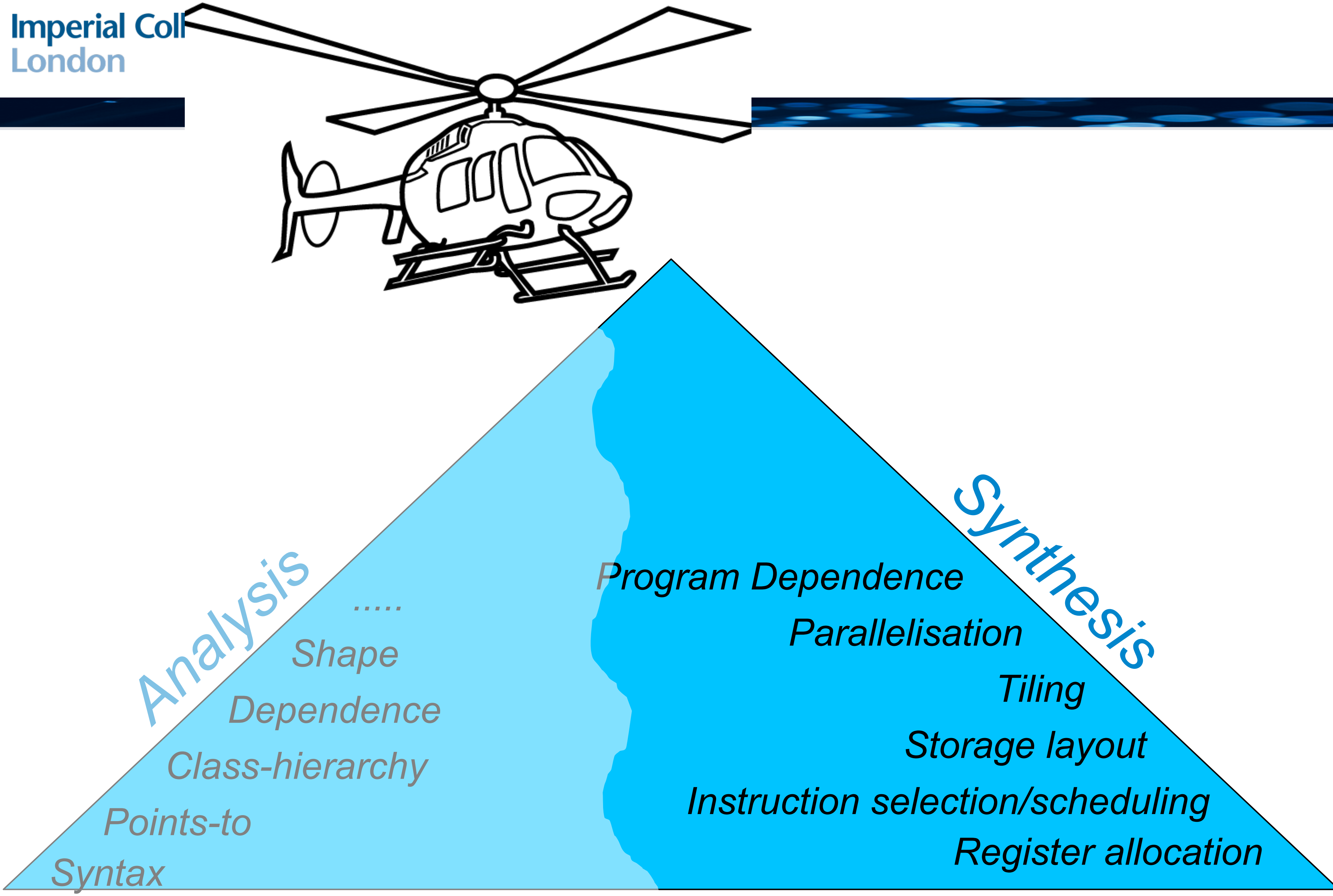
- Domain-specific "active" library encapsulates specialist performance expertise
- Each new platform requires new performance tuning effort
- So domain-specialists will be doing the performance tuning
- Our challenge is to support them



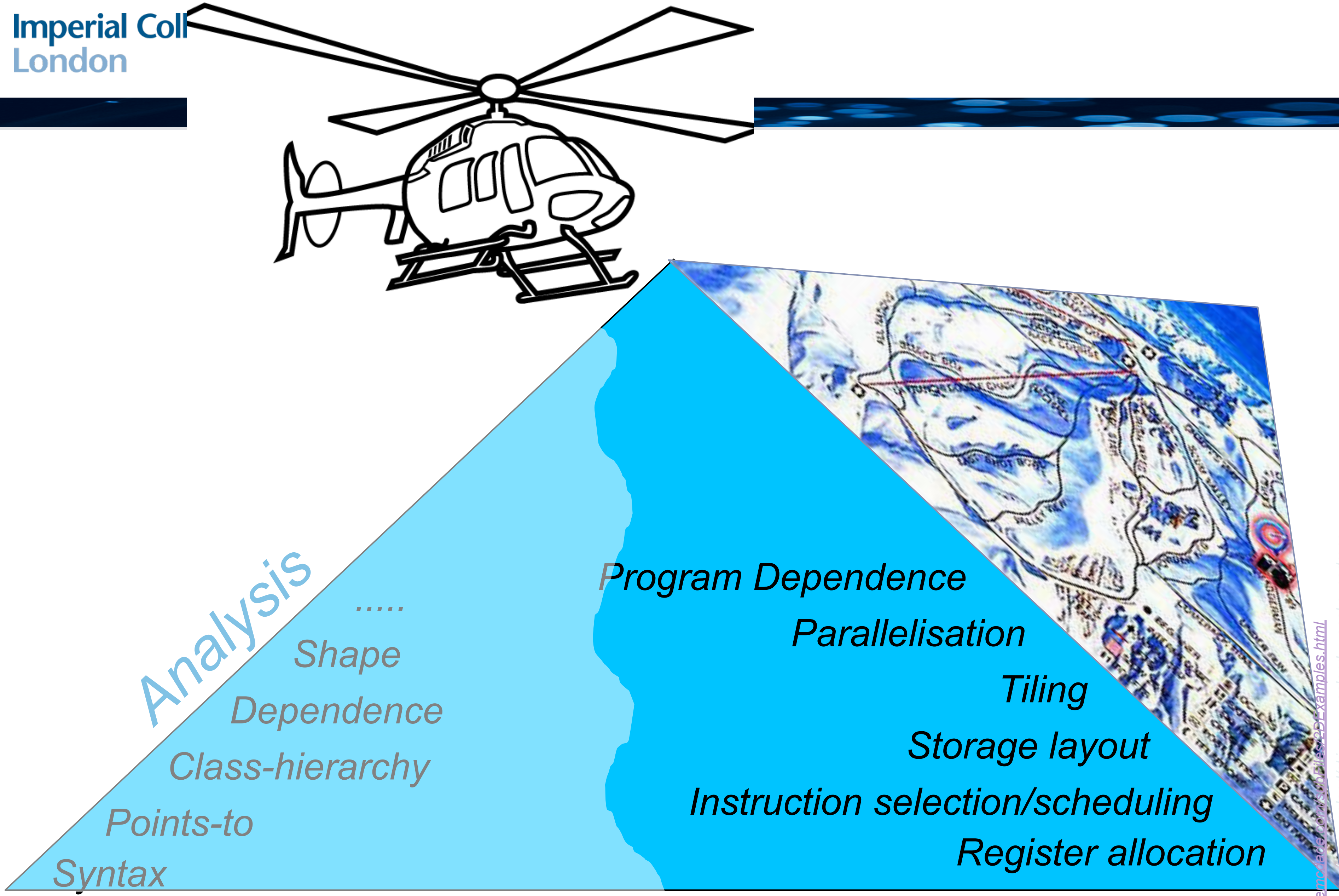


- Classical compilers have two halves

Imperial College London



- The right domain-specific language or active library can give us a free ride



- It turns out that analysis is not always the interesting part....

Imperial College
London

C,C++, C#, Java, Fortran

Code motion optimisations

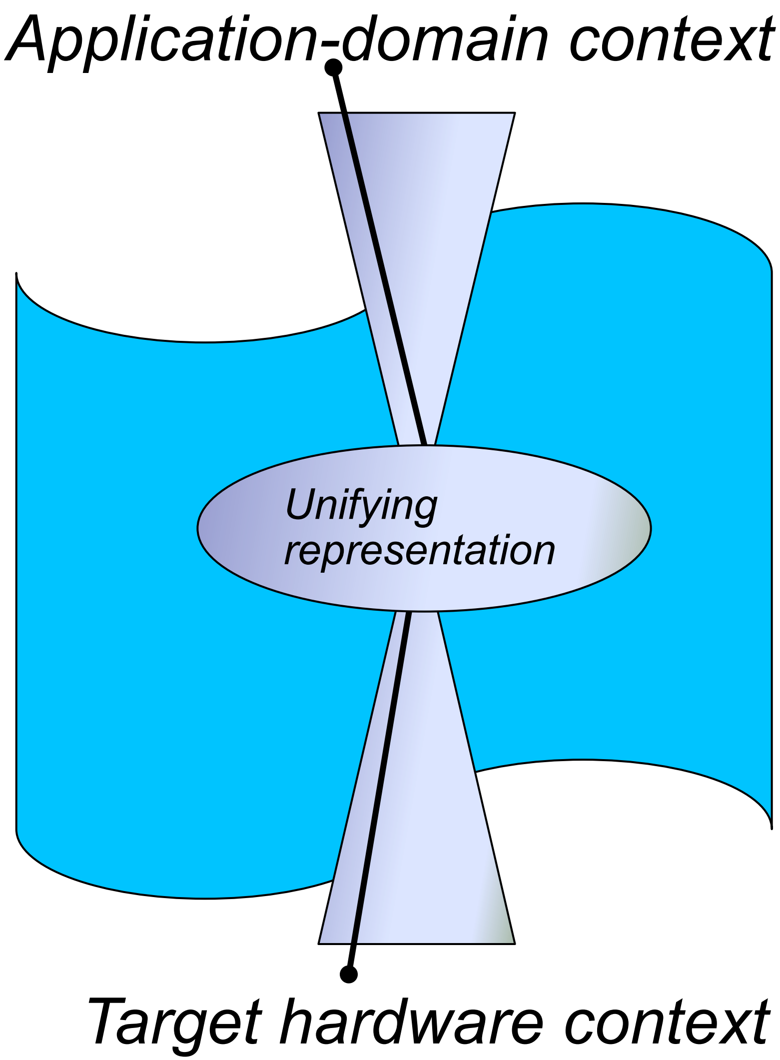
Vectorisation and parallelisation of affine loops over arrays

Capture dependence and communication in programs over richer data structures

Specify application requirements, leaving implementation to select radically-different solution approaches

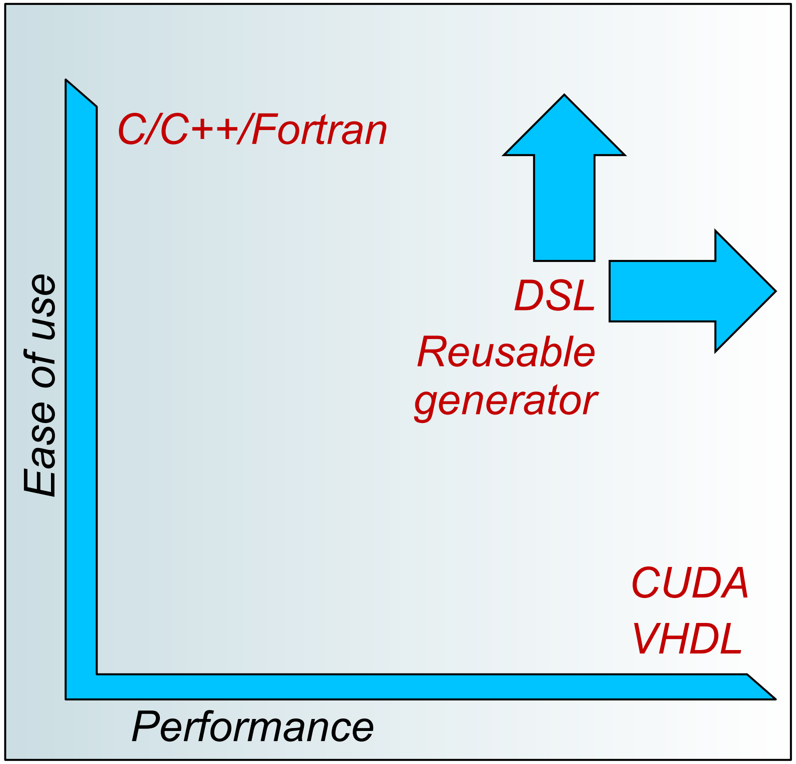
Encapsulating and delivering domain expertise

- Domain-specific languages & active libraries
  - Raise the level of **abstraction**
  - Capture a domain of **variability**
  - Encapsulate **reuse** of a body of code generation expertise/ techniques
- Enable us to capture **design space**
- To match implementation choice to application **context**:
  - Target hardware
  - Problem instance
- This talk illustrates these ideas with some of our recent/current projects



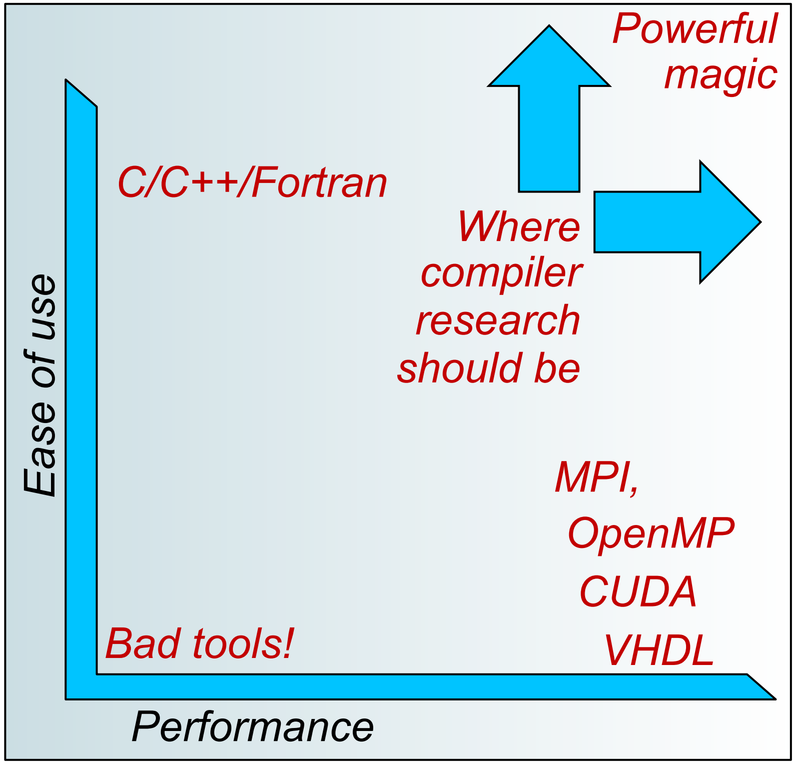
Having your cake and eating it

- If we get this right:
  - Higher performance than you can reasonably achieve by hand
    - the DSL delivers reuse of expert techniques
    - Implements extremely aggressive optimisations
  - Performance portability
    - Isolate long-term value embodied in higher levels of the software from the optimisations needed for each platform
  - Raised level of abstraction
    - Promoting new levels of sophistication
    - Enabling flexibility
  - Domain-level correctness



Having your cake and eating it

- If we get this right:
  - Higher performance than you can reasonably achieve by hand
    - the DSL delivers reuse of expert techniques
    - Implements extremely aggressive optimisations
  - Performance portability
    - Isolate long-term value embodied in higher levels of the software from the optimisations needed for each platform
  - Raised level of abstraction
    - Promoting new levels of sophistication
    - Enabling flexibility
  - Domain-level correctness

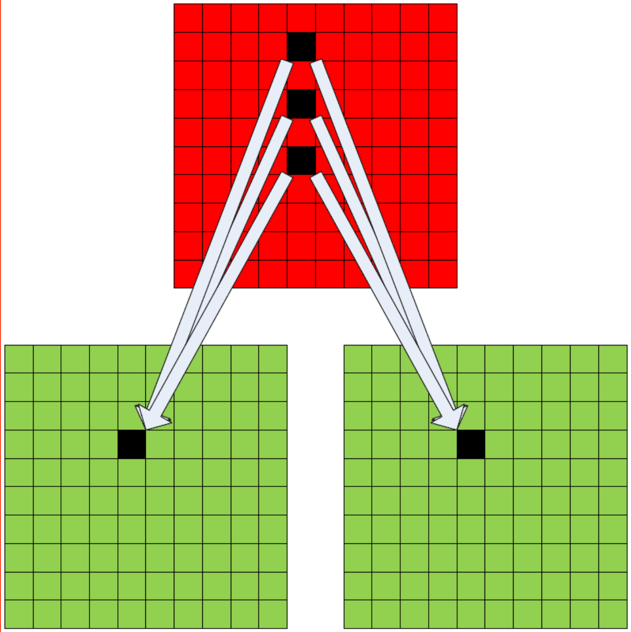


Indexed functor

Domain-specific active library example

- *Functor represents function over an image*
- *Kernel accesses image via indexers*
- *Indexers carry metadata that characterises kernel's data access pattern*

```
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput,  eComponent, e1D> Input;
  Indexer<eOutput, eComponent, e0D> HighOutput;
  Indexer<eOutput, eComponent, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius radius) : Input(axis, radius) {}

  void Kernel() {
    float centre = Input();
    float high = (centre - (Input(-Input.Radius) +
                 Input(Input.Radius)) * 0.5f) * 0.5f;

    HighOutput() = high;
    LowOutput() = centre - high;
  }
};
```



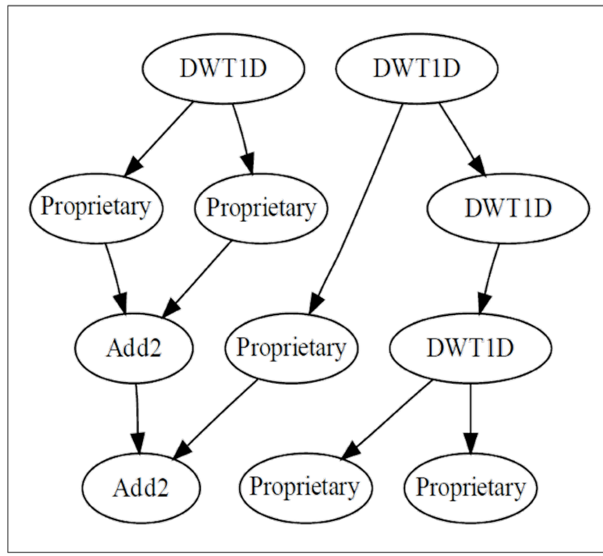
Vertical DWT
(1:2 / Filter Skeleton)

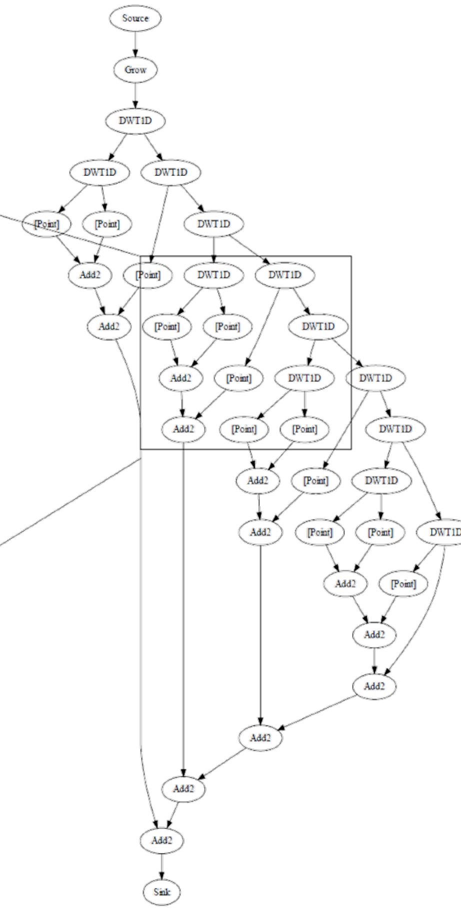
- *One-dimensional discrete wavelet transform, as indexed functor*
- *Compilable with standard C++ compiler*
- *Operates in either the horizontal or vertical axis*
  - *Input indexer operates on RGB components separately*
  - *Input indexer accesses ±radius elements in one (the axis) dimension*

Imperial College London

Image degrain example

```
Image DeGrainRecurse(Image input, int level = 0) {
  Image HY,LY,HH,HL,LH,LL,HHP,HLP,LHP,LLP,pSum1,pSum2,out;

  DWT1D hDWT(eHorizontal, 1 << level);
  DWT1D vDWT(eVertical, 1 << level);
  hDWT(input, HY, LY);
  vDWT(HY, HH, LH);
  vDWT(LY, LH, LL);

  Proprietary prop;
  prop(HH, HHP);
  prop(LH, LHP);
  prop(HL, HLP);

  Sum sum;
  sum(HHP, LHP, pSum1);
  sum(HLP, pSum1, pSum2);

  /* Go to the next level of recursion. */
  LLP = (level < 3) ? DeGrainRecurse(LL, level+1) : LL;

  sum(pSum2, LLP, out);
  return out;
}
```

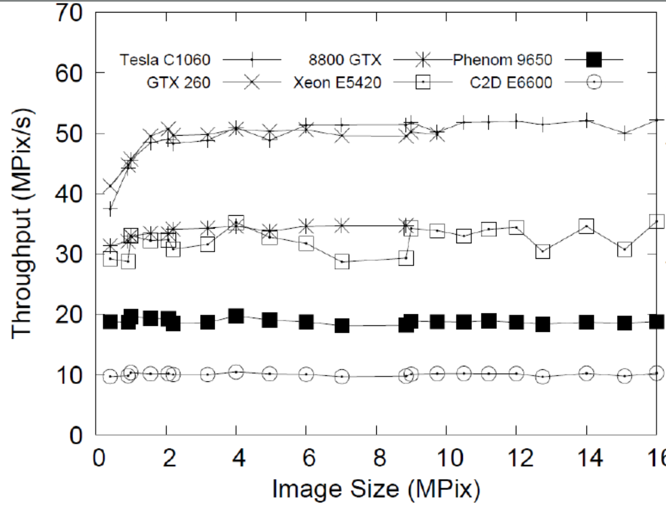




- *Recursive wavelet-based degraining visual effect in C++*
- *Visual primitives are chained together via image temporaries to form a DAG*
- *DAG construction is captured through delayed evaluation.*
  - *Collaboration with The Foundry Ltd, www.thefoundry.co.uk, visual effects for film post-production*

Imperial College London

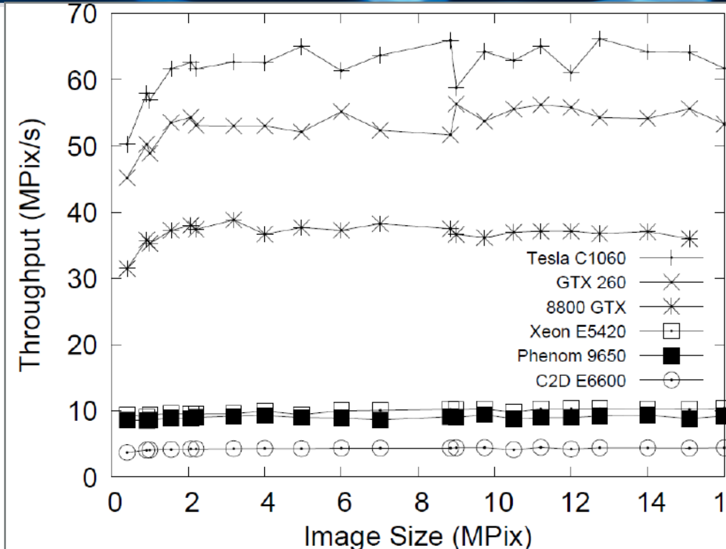
Performance – Multicore +SSE vs NVidia GPUs



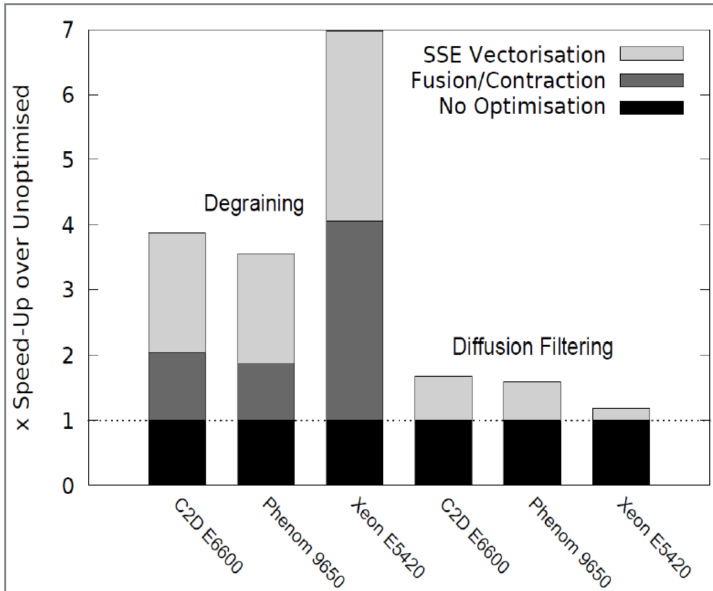
- In this example, CPU *can* beat a GPU
- Because loop fusion eliminates DRAM bottleneck
- Future work: loop fusion for the GPU!

- Tesla C1060 (nVidia)
  - 30-SM, CC 1.3
- GTX 260 (nVidia)
  - 24-SM, CC 1.3
- 8800 GTX (nVidia)
  - 16-SM, CC 1.0
- Phenom 9650 (AMD)
  - 4-core
- Xeon E5420 (Intel)
  - 8 cores, two sockets, two Core2Duos per socket
- C2D E6600 (Intel)
  - 2-Core Core2Duo

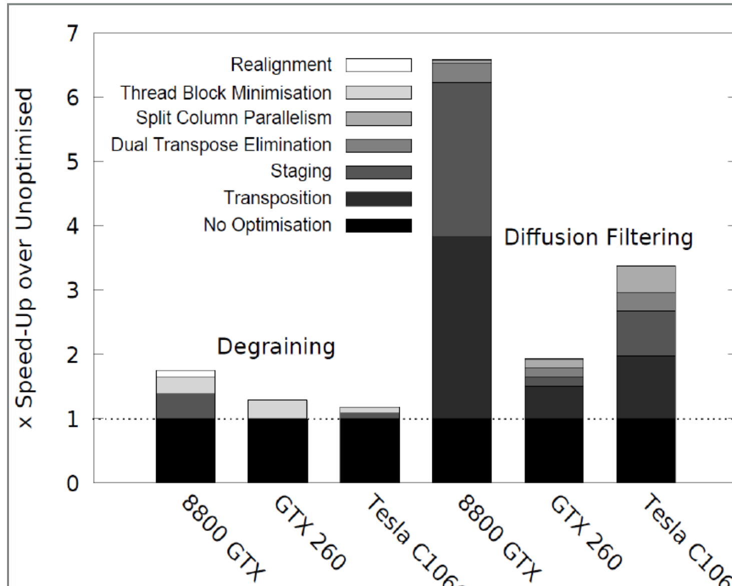
- All systems ran 64-bit Ubuntu Linux 8 with the Intel C/C++ Compiler 11.0, CUDA Toolkit 2.1 and 180 series NVIDIA graphics drivers.We used ICC flags "-O3 –xHost no-prec-div -ansi-alias -restrict" and NVCC flag "-O3".
- GPU timings do not include host/device data transfers.
- images were stock photos cropped or repeated to a set of industry-standard rame sizes, powers-of-two and prime numbers



- In this example, GPUs *always* win
- Loop fusion is not possible
- So GPU DRAM bandwidth gives overwhelming advantage
- 8 cores are no better than 4 cores since bandwidth-limited



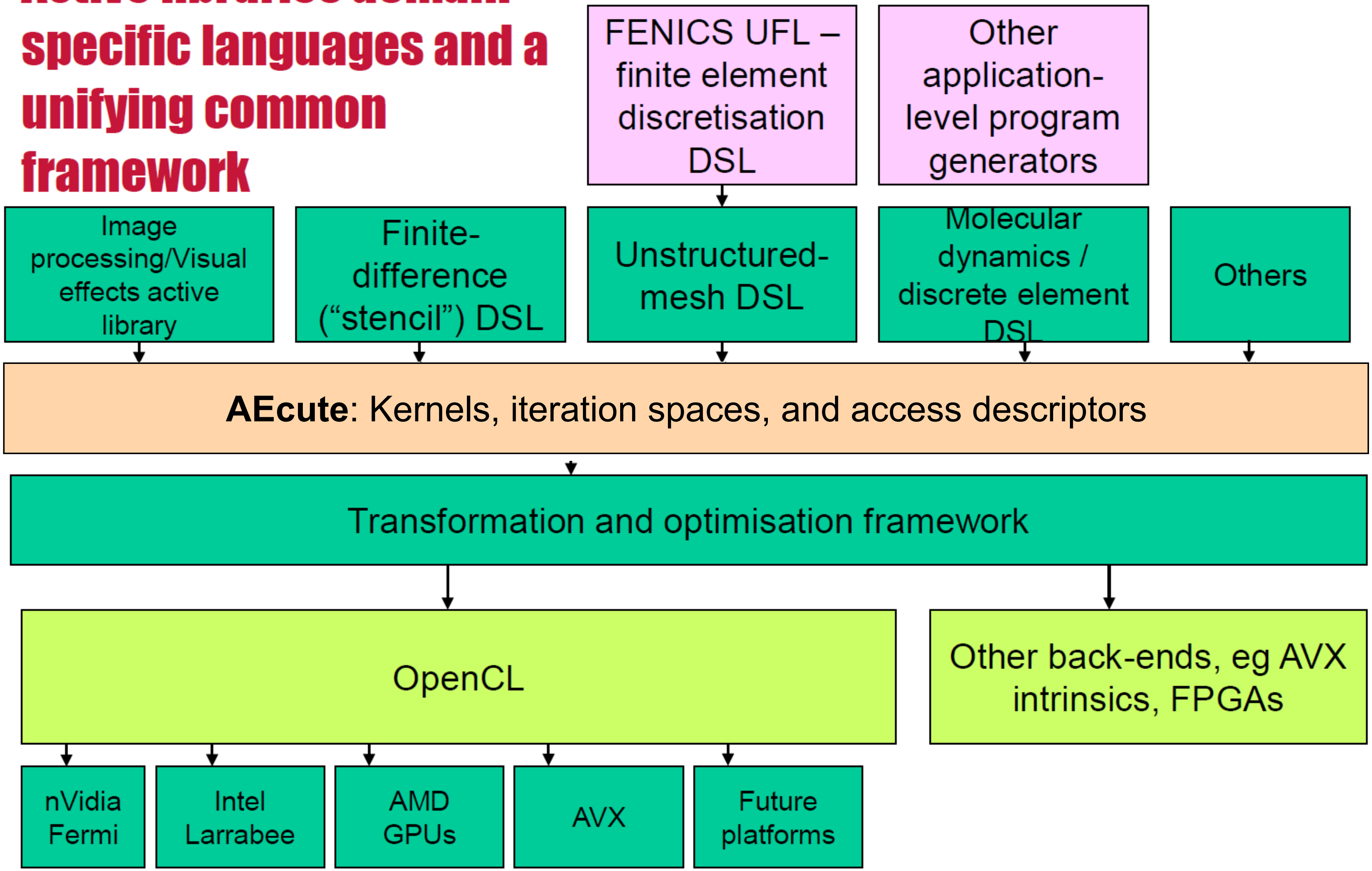
- Without loop fusion, SSE is of limited value – memory is bottleneck
- 8-core Intel Xeon has less DRAM and L2 bandwidth per core, so benefits more from fusion



- Older nVidia hardware was very sensitive to alignment of global memory accesses – not a problem with GTX260 and C1060
- Staging and transposition are crucial for diffusion filtering

This research prototype is part of the foundation for The Foundry's forthcoming BLINK developer tool

Active libraries domain-specific languages and a unifying common framework



Active libraries domain-specific languages and a unifying common framework

FENICS UFL – finite element discretisation DSL

Other application-level program generators

Image processing/Visual effects active library

Finite-difference (“stencil”) DSL

Unstructured-mesh DSL

Molecular dynamics / discrete element DSL

Others

What does this have to do?

- **AEcute**: Kernels, iteration spaces, and access descriptors

Transformation and optimisation framework

OpenCL

Other back-ends, eg AVX intrinsics, FPGAs

nVidia Fermi

Intel Larrabee

AMD GPUs

AVX

Future platforms

Active libraries domain-specific languages and a unifying common framework

FENICS UFL – finite element discretisation DSL

Other application-level program generators

Image processing/Visual effects active library

Finite-difference (“stencil”) DSL

Unstructured-mesh DSL

Molecular dynamics / discrete element DSL

Others

•**AEcute**: Kernels, iteration spaces, and access descriptors

- *Automate synthesis of data movement code*
- *Automatically partition and parallelise*
- *Automatically select storage layouts and schedules to maximise spatial locality and alignment*
- *Automatically fuse loops and contract intermediate arrays*

Transformation and optimisation framework

OpenCL

Other back-ends, eg AVX intrinsics, FPGAs

nVidia Fermi

Intel Larrabee

AMD GPUs

AVX

Future platforms

Active libraries domain-specific languages and a unifying common framework

FENICS UFL – finite element discretisation DSL

Other application-level program generators

Image processing/Visual effects active library

Finite-difference ("stencil") DSL

Unstructured-mesh DSL

Molecular dynamics / discrete element DSL

Others

- *Explicitly characterise what data will be accessed*
- *At each point in the kernel's iteration space*
- *As a function of its position*

• **AEcute**: Kernels, iteration spaces, and access descriptors

- *Automate synthesis of data movement code*
- *Automatically partition and parallelise*
- *Automatically select storage layouts and schedules to maximise spatial locality and alignment*
- *Automatically fuse loops and contract intermediate arrays*

Transformation and optimisation framework

OpenCL

Other back-ends, eg AVX intrinsics, FPGAs

nVidia Fermi

Intel Larrabee

AMD GPUs

AVX

Future platforms

Imperial College London

Active libraries domain-specific languages and a unifying common framework

FENICS UFL – finite element discretisation DSL | Other application-level program generators

Image processing/Visual effects active library | Finite-difference ("stencil") DSL | Unstructured-mesh DSL | Molecular dynamics / discrete element DSL | Others

- Explicitly characterise what data will be accessed
- At each point in the kernel's iteration space
- As a function of its position

- **Idea: decoupling**

Access operands

Execute

Write back

(the essence of streaming)

(generalised)

- Automate synthesis of data movement code
- Automatically partition and parallelise
- Automatically select storage layouts and schedules to maximise spatial locality and alignment
- Automatically fuse loops and contract intermediate arrays

Transformation and optimisation framework

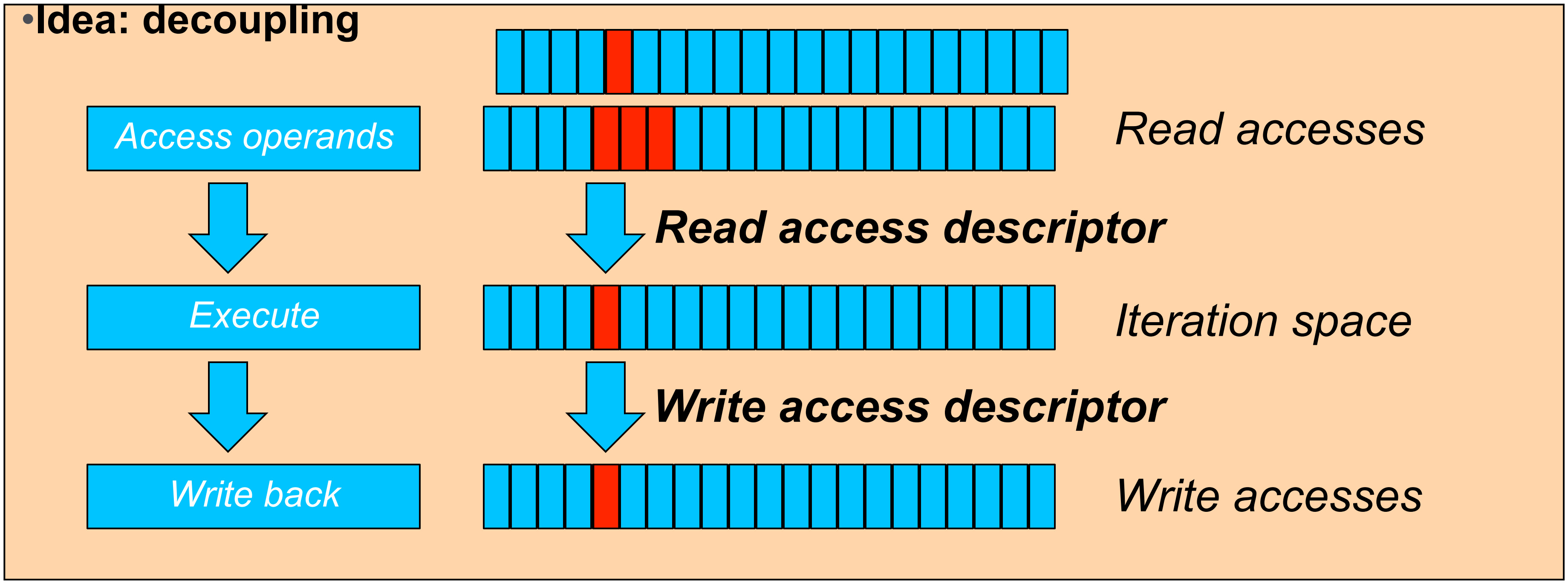
OpenCL | Other back-ends, eg AVX intrinsics, FPGAs

nVidia Fermi | Intel Larrabee | AMD GPUs | AVX | Future platforms

The AEcute programming model

AEcute

- *Decoupled Access/Execute descriptors*



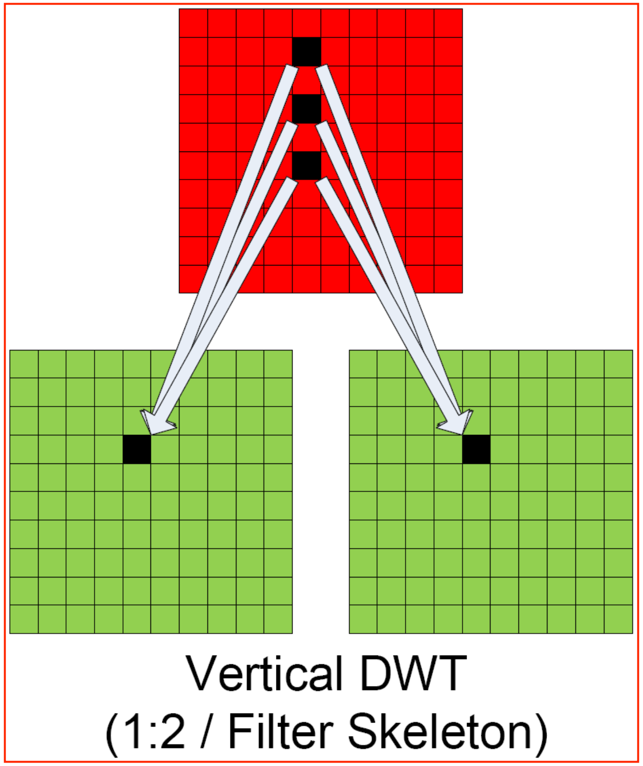
- *Explicitly characterise what data will be accessed*
- *At each point in the kernel's iteration space*
- *As a function of its position*

AEcute and Indexed functors

- ***The "indexed functors" from our visual effects framework are an instance of the AEcute idea***

```
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput,   eComponent, e1D> Input;
  Indexer<eOutput,  eComponent, e0D> HighOutput;
  Indexer<eOutput,  eComponent, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius radius) : Input(axis, radius) {}

  void Kernel() {
    float centre = Input();
    float high = (centre - (Input(-Input.Radius) +
                 Input(Input.Radius)) * 0.5f) * 0.5f;

    HighOutput() = high;
    LowOutput() = centre - high;
  }
};
```

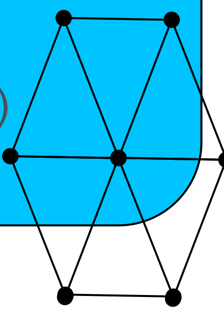
Kernel ***Indexers*** ***Access descriptors***



OP2 – a decoupled access-execute active library for unstructured mesh computations

```
// declare sets, maps, and datasets
op_set nodes = op_decl_set( nnodes );
op_set edges = op_decl_set( nedges );

op_map pedge1 = op_decl_map (edges,
      nodes, 1, mapData1 );
op_map pedge2 = op_decl_map (edges,
      nodes, 1, mapData2 );

op_dat p_A = op_decl_dat (edges, 1, A );
op_dat p_r = op_decl_dat (nodes, 1, r );
op_dat p_u  = op_decl_dat (nodes, 1, u );
op_dat p_du = op_decl_dat (nodes, 1, du );

// global variables and constants declarations
float alpha[2] = { 1.0f, 1.0f };
op_decl_const ( 2, alpha );
```

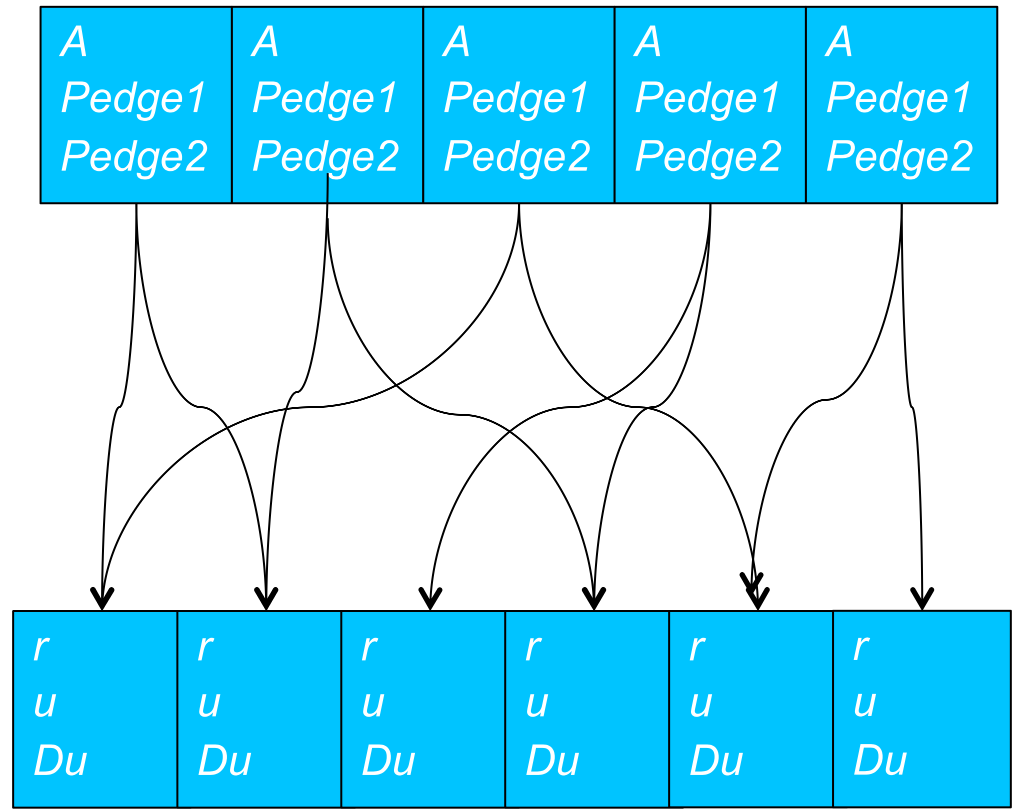
```
float u_sum, u_max, beta = 1.0f;

for ( int iter = 0; iter < NITER; iter++ )
{  op_par_loop ( res, edges,
    op_arg_dat ( p_A,  0, NULL,  OP_READ ),
    op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
    op_arg_dat ( p_du, 0, &pedge1, OP_INC  ),
    op_arg_gbl ( &beta, OP_READ )
    );
   u_sum = 0.0f; u_max = 0.0f;
   op_par_loop ( update, nodes,
    op_arg_dat ( p_r,  0, NULL, OP_READ ),
    op_arg_dat ( p_du, 0, NULL, OP_RW ),
    op_arg_dat ( p_u, 0, NULL, OP_INC ),
    op_arg_gbl ( &u_sum, OP_INC ),
    op_arg_gbl ( &u_max, OP_MAX )
    );
}
```



Example – Jacobi solver

OP2- Data model

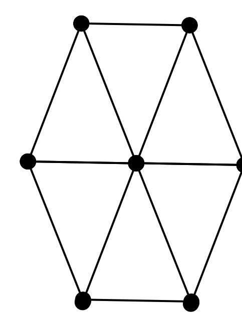
```
// declare sets, maps, and datasets
op_set nodes = op_decl_set( nnodes );
op_set edges = op_decl_set( nedges );

op_map pedge1 = op_decl_map (edges,
     nodes, 1, mapData1 );
op_map pedge2 = op_decl_map (edges,
     nodes, 1, mapData2 );

op_dat p_A = op_decl_dat (edges, 1, A );
op_dat p_r = op_decl_dat (nodes, 1, r );
op_dat p_u  = op_decl_dat (nodes, 1, u );
op_dat p_du = op_decl_dat (nodes, 1, du );

// global variables and constants declarations
float alpha[2] = { 1.0f, 1.0f };
op_decl_const ( 2, alpha );
```



OP2's key data structure is a set

A set may contain pointers that map into another set

Eg each edge points to two vertices



OP2 – a decoupled access-execute active library for unstructured mesh computations

- Each parallel loop precisely characterises the data that will be accessed by each iteration
- This allows staging into scratchpad memory
- And gives us precise dependence information
- In this example, the “res” kernel visits each edge
  - reads edge data, A
  - Reads beta (a global),
  - Reads u belonging to the vertex pointed to by “edge2”
  - Increments du belonging to the vertex pointed to by “edge1”

```
float u_sum, u_max, beta = 1.0f;

for ( int iter = 0; iter < NITER; iter++ )
{
  op_par_loop_4 ( res, edges,
    op_arg_dat ( p_A,  0, NULL,  OP_READ ),
    op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
    op_arg_dat ( p_du, 0, &pedge1, OP_INC  ),
    op_arg_gbl ( &beta, OP_READ )
  );
  u_sum = 0.0f; u_max = 0.0f;
  op_par_loop_5 ( update, nodes,
    op_arg_dat ( p_r,  0, NULL, OP_READ ),
    op_arg_dat ( p_du, 0, NULL, OP_RW ),
    op_arg_dat ( p_u, 0, NULL, OP_INC ),
    op_arg_gbl ( &u_sum, OP_INC ),
    op_arg_gbl ( &u_max, OP_MAX )
  );
}
```

Example – Jacobi solver

OP2 – parallel loops

```
inline void res(const float A[1], const float u[1],
                float du[1], const float beta[1])
{
  du[0] += beta[0]*A[0]*u[0];
}
```

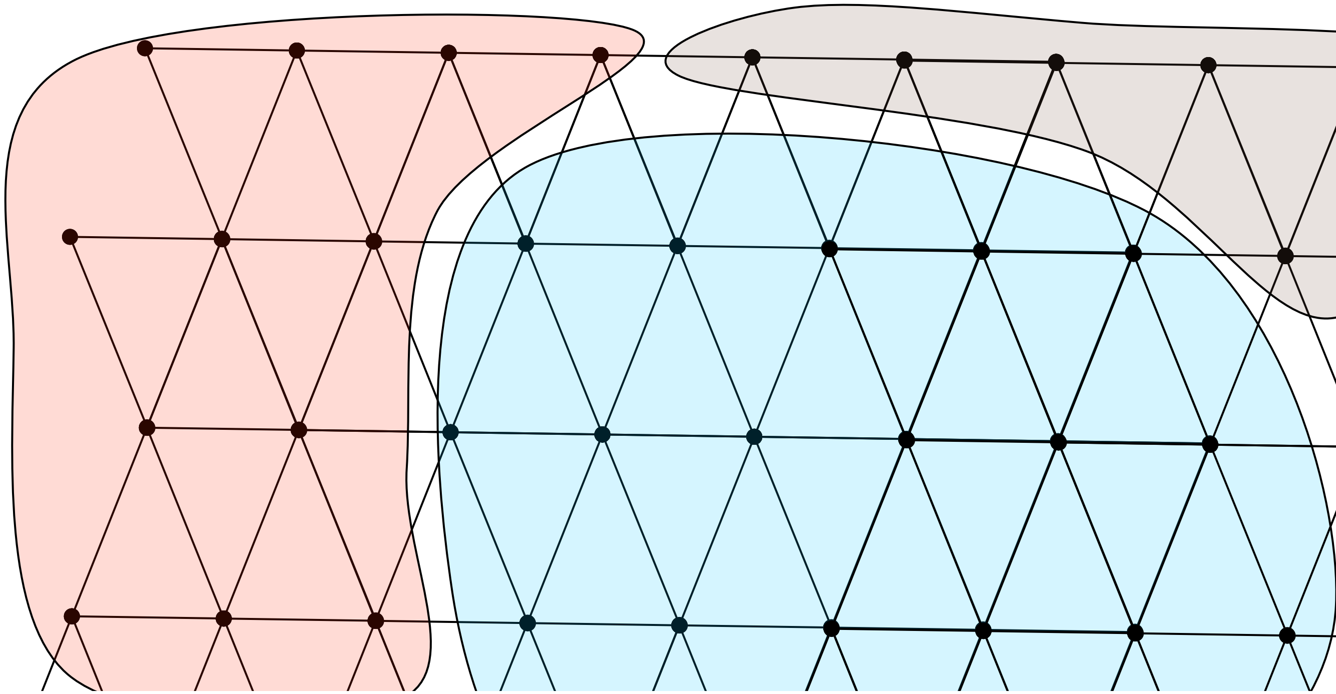
```
inline void update(const float r[1], float du[1],
      float u[1], float u_sum[1], float u_max[1])
{
  u[0] += du[0] + alpha * r[0];
  du[0] = 0.0f;
  u_sum[0] += u[0]*u[0];
  u_max[0] = MAX(u_max[0],u[0]);
}
```

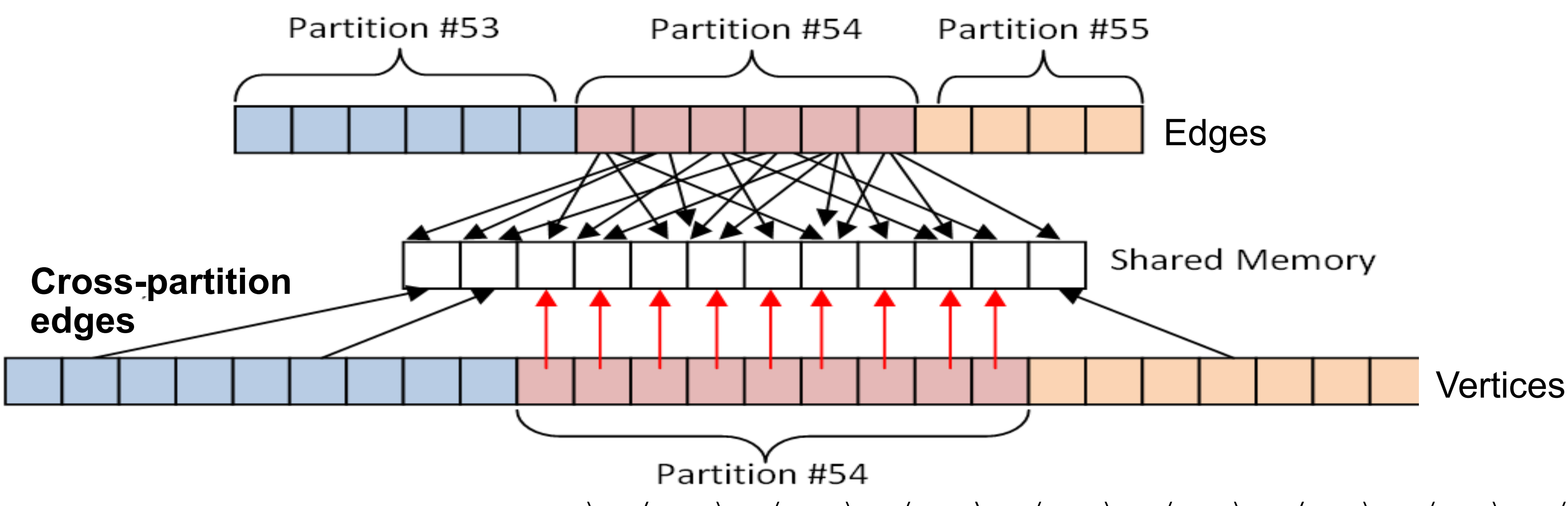
```
float u_sum, u_max, beta = 1.0f;

for ( int iter = 0; iter < NITER; iter++ )
{
  op_par_loop_4 ( res, edges,
    op_arg_dat ( p_A,  0, NULL,  OP_READ ),
    op_arg_dat ( p_u, 0, &pedge2, OP_READ ),
    op_arg_dat ( p_du, 0, &pedge1, OP_INC  ),
    op_arg_gbl ( &beta, OP_READ )
    );
  u_sum = 0.0f; u_max = 0.0f;
  op_par_loop_5 ( update, nodes,
    op_arg_dat ( p_r,  0, NULL, OP_READ ),
    op_arg_dat ( p_du, 0, NULL, OP_RW ),
    op_arg_dat ( p_u, 0, NULL, OP_INC ),
    op_arg_gbl ( &u_sum, OP_INC ),
    op_arg_gbl ( &u_max, OP_MAX )
    );
}
```

- In this example, the "res" kernel visits each edge
  - reads edge data, A
  - Reads beta (a global),
  - Reads u belonging to the vertex pointed to by "edge2"
  - Increments du belonging to the vertex pointed to by "edge1"

Example – Jacobi solver

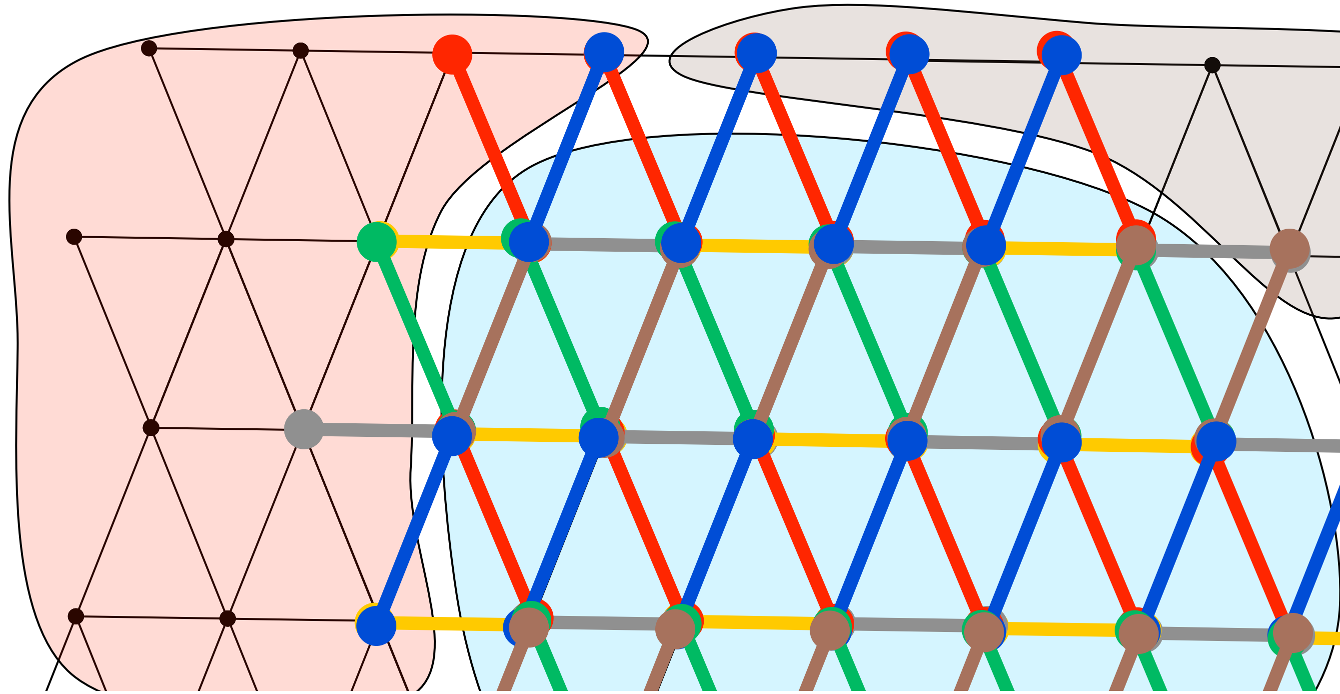
- Two key optimisations:
- **Partitioning**
- Colouring

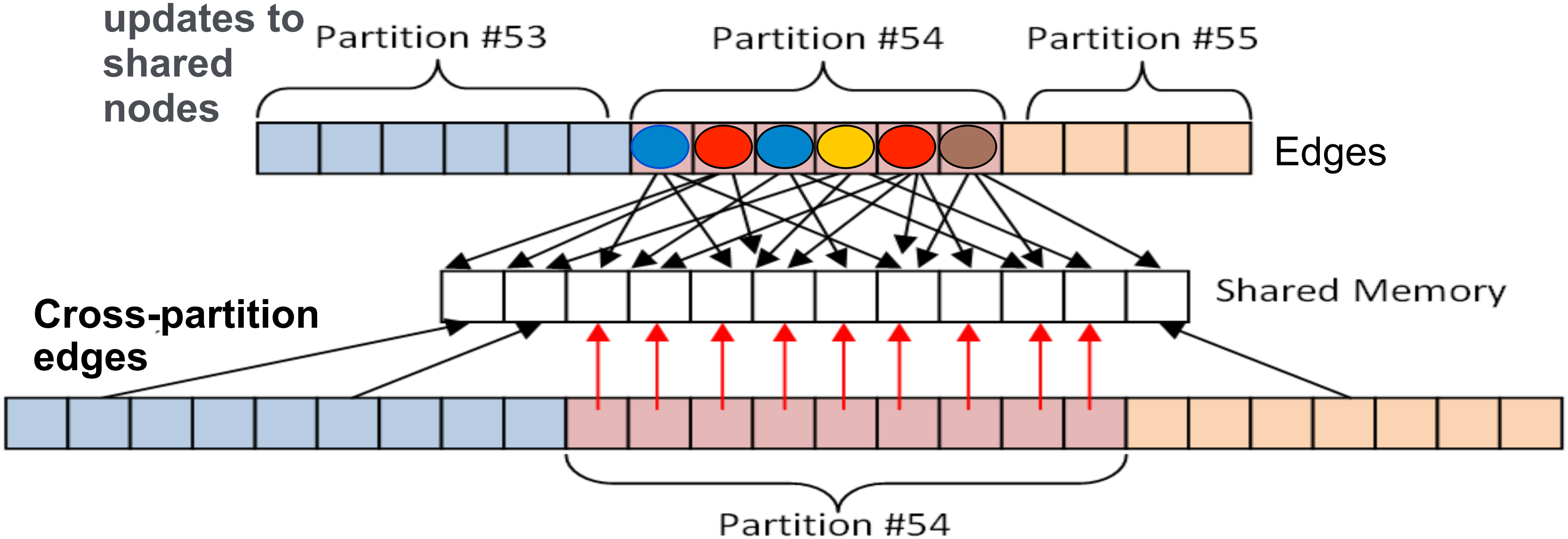




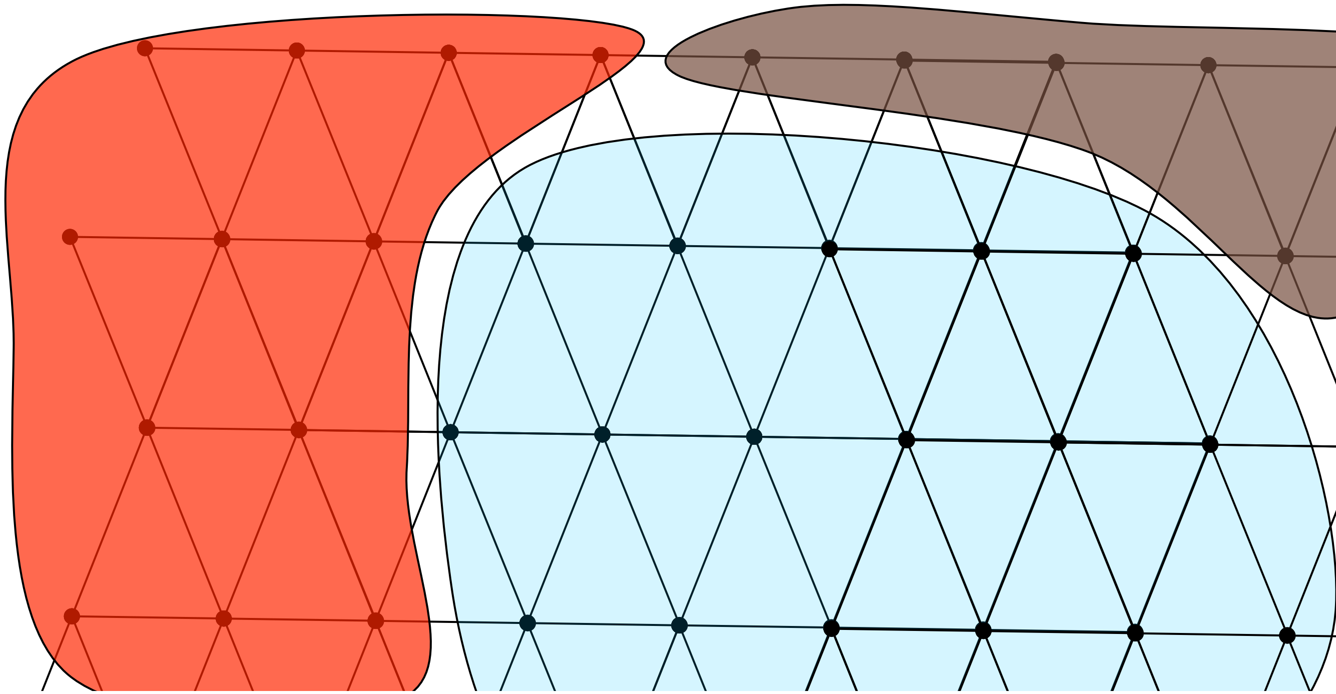
- Two key optimisations:

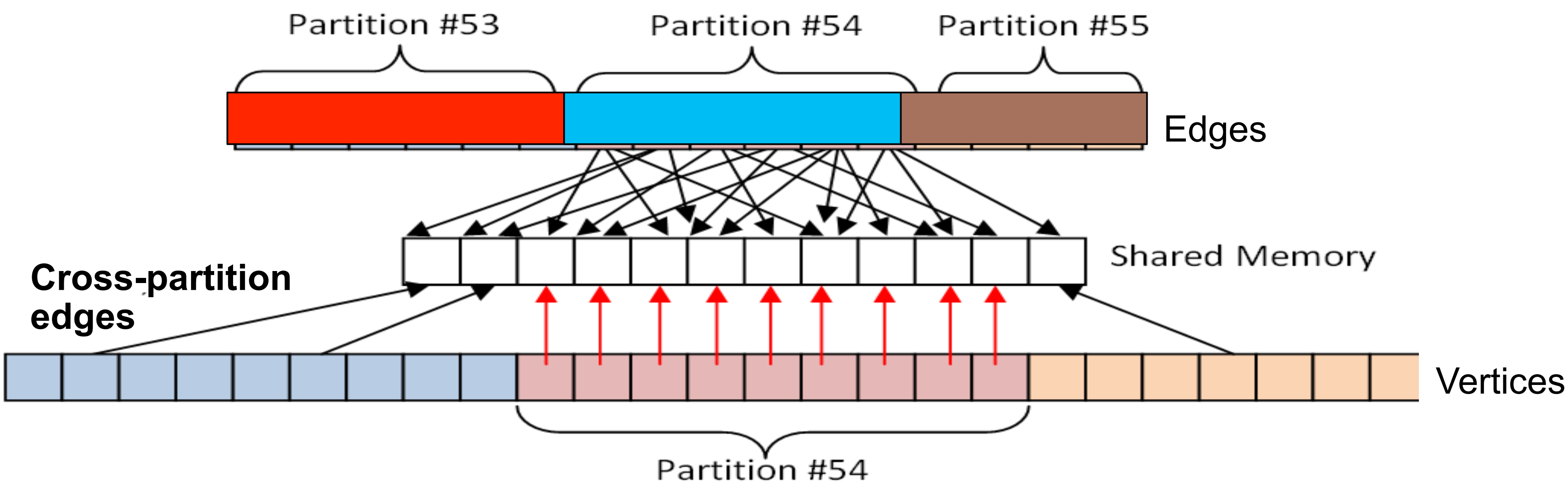
- Partitioning
- **Colouring**
  - **Elements of the edge set are coloured to avoid races due to concurrent updates to shared nodes**





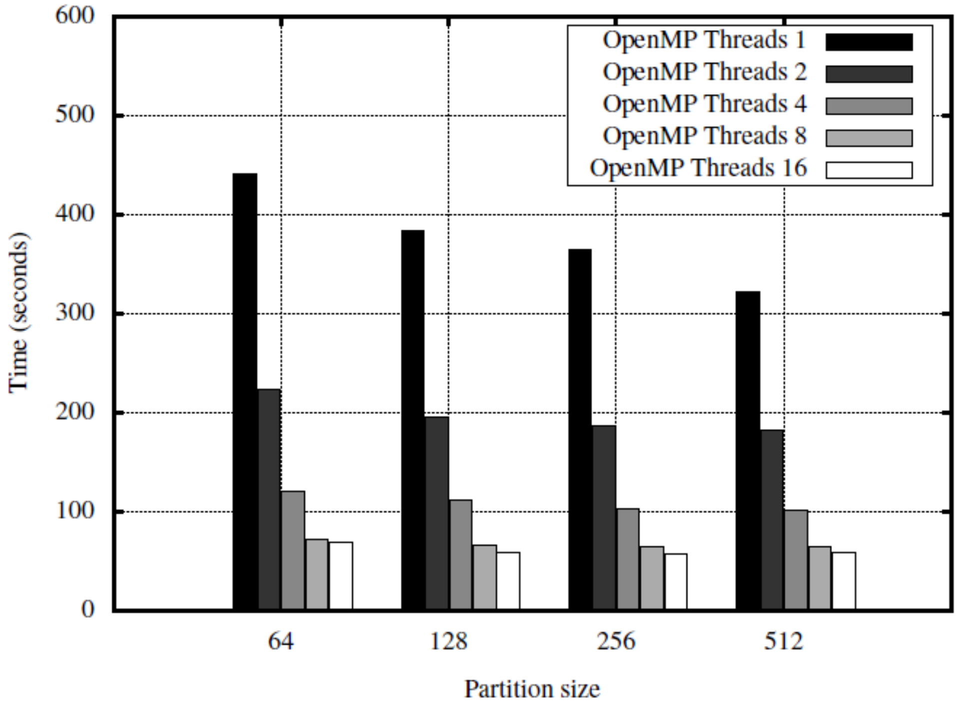
- Two key optimisations:
- **Partitioning**
- Colouring
  - At two levels



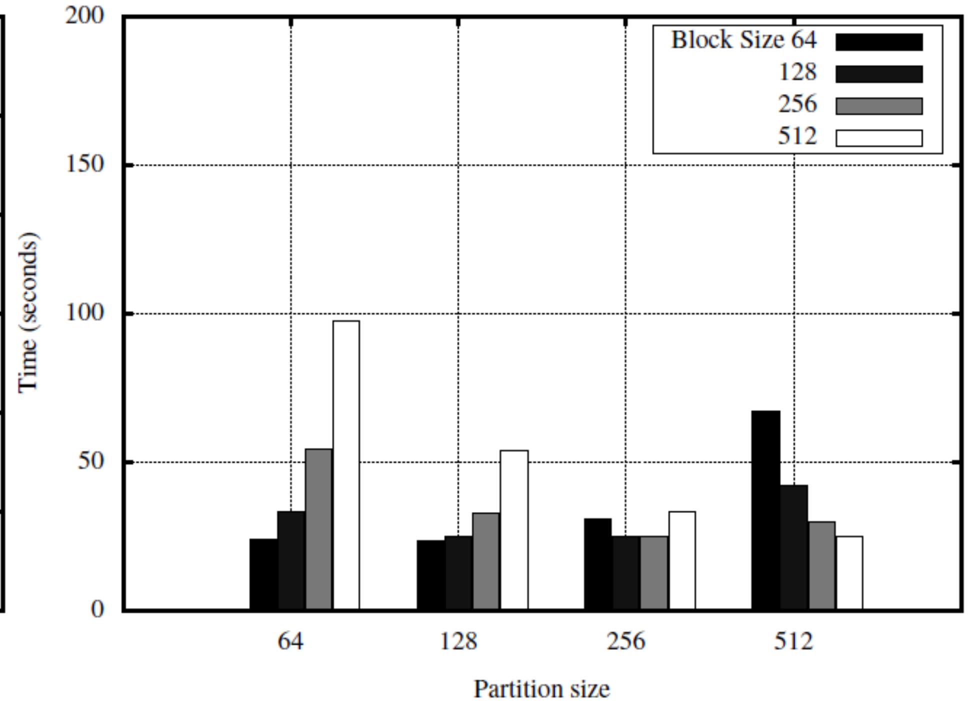


Imperial College London

OP2 - performance

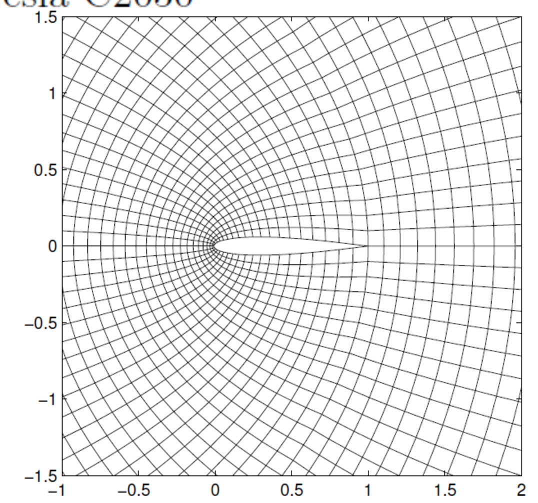


(a) Intel Xeon E5540 (Nehalem) (ICC 11.1)



(b) Tesla C2050

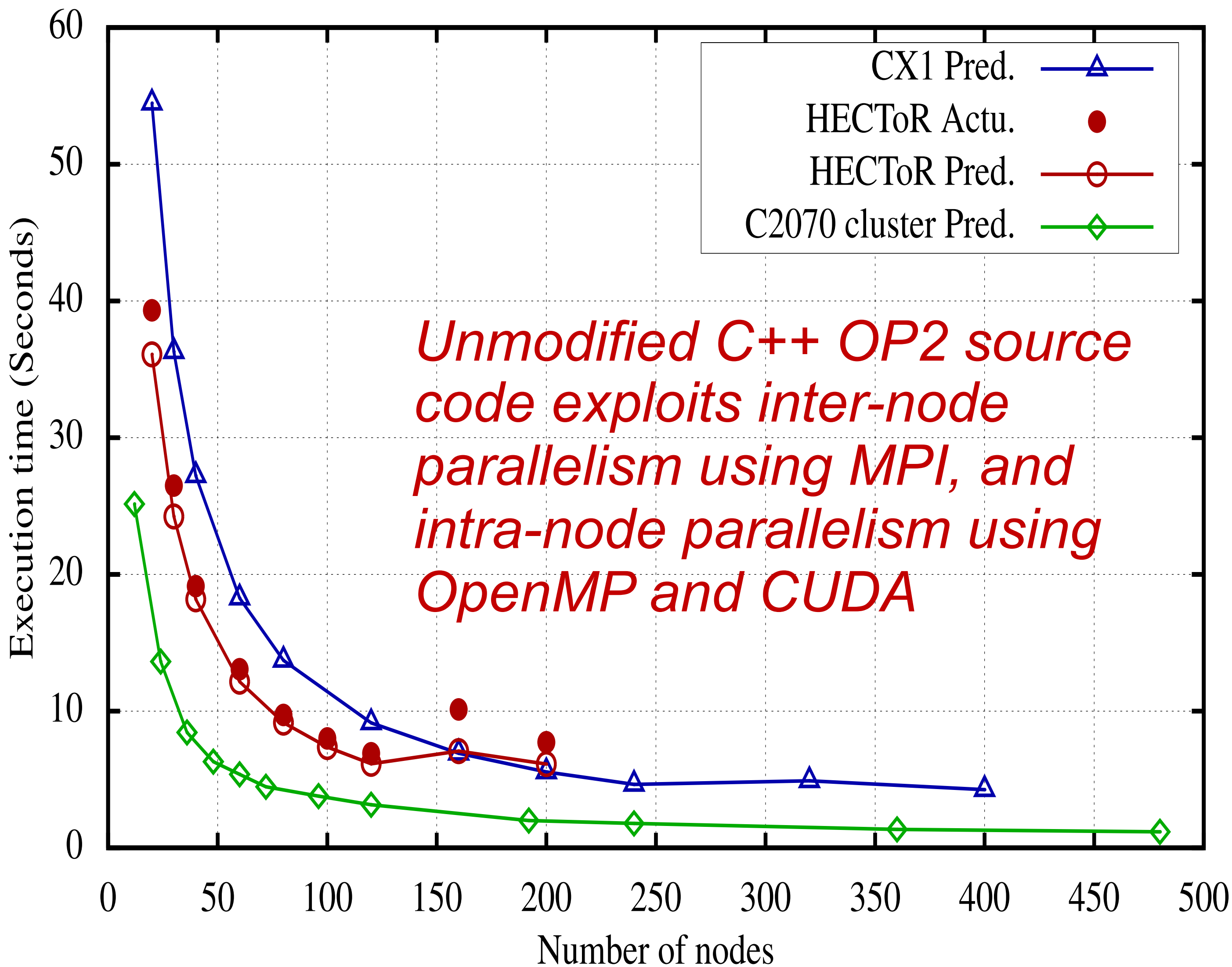
- Example: non-linear 2D inviscid unstructured airfoil code, double precision (compute-light, data-heavy)
- Two backends: OpenMP, CUDA (OpenCL coming)
- For tough, unstructured problems like this GPUs can win, but you have to work at it
- X86 also benefits from tiling; we are looking at how to enhance SSE/AVX exploitation



Imperial College London

Combining MPI, OpenMP and CUDA

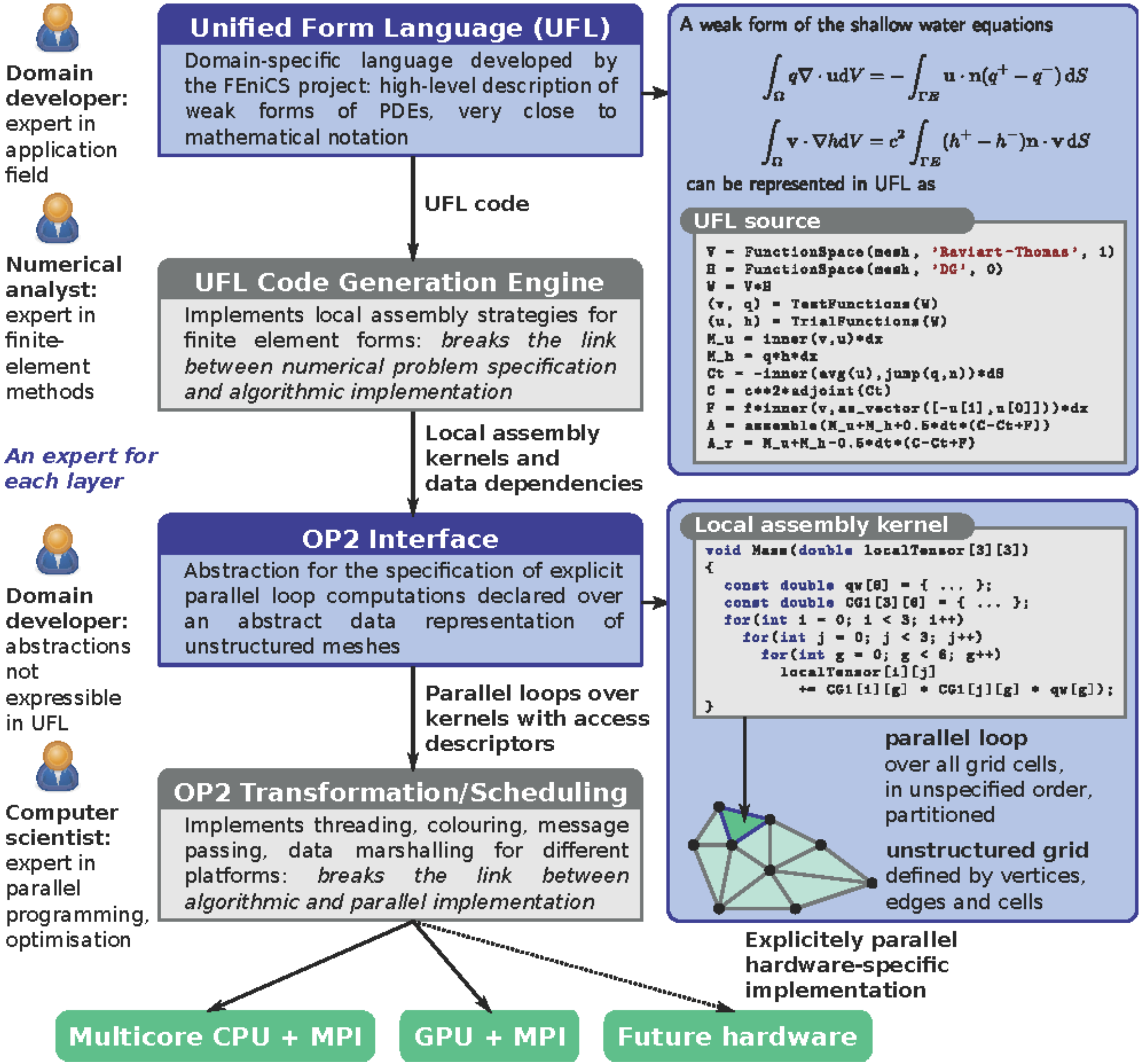
- *non-linear 2D inviscid airfoil code*
- *26M-edge unstructured mesh*
- *1000 iterations*
- *Analytical model validated on up to 120 Westmere X5650 cores and 1920 HECToR (Cray XE6) cores*



(Preliminary results under review)

What does a DSL give you?

- Semantic properties deriving from the domain-level
  - example: SPIRAL's rewrite rules for decomposing linear transforms
- Simplified reasoning deriving from operating at a higher level of representation
  - example: SPIRAL but also DESOLA's treatment of fusion of loops over sparse matrices
- Delivering optimisations and implementation techniques specifically known to be valuable for a class of applications
  - example: OP2's partitioning, staging and colouring schemes for indirect loops over unstructured meshes
- Opening-up the design space, so that we can freely navigate to the optimum implementation technique for each application context and each hardware platform.



Domain developer: expert in application field
Numerical analyst: expert in finite-element methods
An expert for each layer
Domain developer: abstractions not expressible in UFL
Computer scientist: expert in parallel programming, optimisation
Unified Form Language (UFL)
Domain-specific language developed by the FEniCS project: high-level description of weak forms of PDEs, very close to mathematical notation
UFL code
UFL Code Generation Engine
Implements local assembly strategies for finite element forms: *breaks the link between numerical problem specification and algorithmic implementation*
Local assembly kernels and data dependencies
OP2 Interface
Abstraction for the specification of explicit parallel loop computations declared over an abstract data representation of unstructured meshes
Parallel loops over kernels with access descriptors
OP2 Transformation/Scheduling
Implements threading, colouring, message passing, data marshalling for different platforms: *breaks the link between algorithmic and parallel implementation*
Multicore CPU + MPI
GPU + MPI
Future hardware
Explicitely parallel hardware-specific implementation
A weak form of the shallow water equations
$$\int_\Omega q\nabla\cdot\mathbf{u}\,dV = -\int_{\Gamma E}\mathbf{u}\cdot\mathbf{n}(q^+ - q^-)\,dS$$
$$\int_\Omega \mathbf{v}\cdot\nabla h\,dV = c^2\int_{\Gamma E}(h^+ - h^-)\mathbf{n}\cdot\mathbf{v}\,dS$$
can be represented in UFL as
UFL source
V = FunctionSpace(mesh, 'Raviart-Thomas', 1)
H = FunctionSpace(mesh, 'DG', 0)
W = V*H
(v, q) = TestFunctions(W)
(u, h) = TrialFunctions(W)
M_u = inner(v,u)*dx
M_h = q*h*dx
Ct = -inner(avg(u),jump(q,n))*dS
C = c**2*adjoint(Ct)
F = f*inner(v,as_vector([-u[1],u[0]]))*dx
A = assemble(M_u+M_h+0.5*dt*(C-Ct+F))
A_r = M_u+M_h-0.5*dt*(C-Ct+F)
Local assembly kernel
void Mass(double localTensor[3][3])
{
const double qw[6] = { ... };
const double CG1[3][6] = { ... };
for(int i = 0; i < 3; i++)
for(int j = 0; j < 3; j++)
for(int g = 0; g < 6; g++)
localTensor[i][j]
+= CG1[i][g] * CG1[j][g] * qw[g]);
}
parallel loop over all grid cells, in unspecified order, partitioned
unstructured grid defined by vertices, edges and cells

	$n_f = 1$				$n_f = 2$				$n_f = 3$				$n_f = 4$			
	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E	Q	T	E	B/E
$p = 1, q = 1$	218	27	28	0.96	260	80	70	**1.14**	350	267	121	**2.20**	679	751	215	**3.15**
$p = 1, q = 2$	820	76	89	0.85	1483	193	160	**1.20**	2092	651	284	**2.29**	3432	1949	501	**3.89**
$p = 1, q = 3$	4946	126	161	0.78	7915	490	410	**1.19**	8057	1559	922	**1.69**	11851	3123	1205	**2.59**
$p = 1, q = 4$	17316	435	485	0.89	24915	1111	1048	**1.06**	25331	2542	2024	**1.25**	34526	4159	2797	**1.48**
$p = 2, q = 1$	253	49	55	0.89	655	315	218	**1.44**	1690	1970	941	**1.79**	2896	10637	2421	**1.19**
$p = 2, q = 2$	1533	117	134	0.87	3424	998	584	**1.70**	5339	5899	2372	**2.25**	-	-	-	-
$p = 2, q = 3$	7857	318	356	0.89	11779	1966	1431	**1.37**	16690	7860	4732	**1.66**	-	-	-	-
$p = 2, q = 4$	24930	853	979	0.87	34435	4306	3603	**1.19**	-							
$p = 3, q = 1$	356	106	90	**1.17**	1767	1023	501	**2.04**	-							
$p = 3, q = 2$	2122	223	217	**1.02**	5443	2743	1473	**1.86**	-							
$p = 3, q = 3$	8113	756	838	0.90	16927	5684	4552	**1.24**	-							
$p = 3, q = 4$	25165	1661	2006	0.82	46034	9856	9746	**1.01**	-							

Evaluation of variational forms involves hard-to-exploit redundant subexpressions

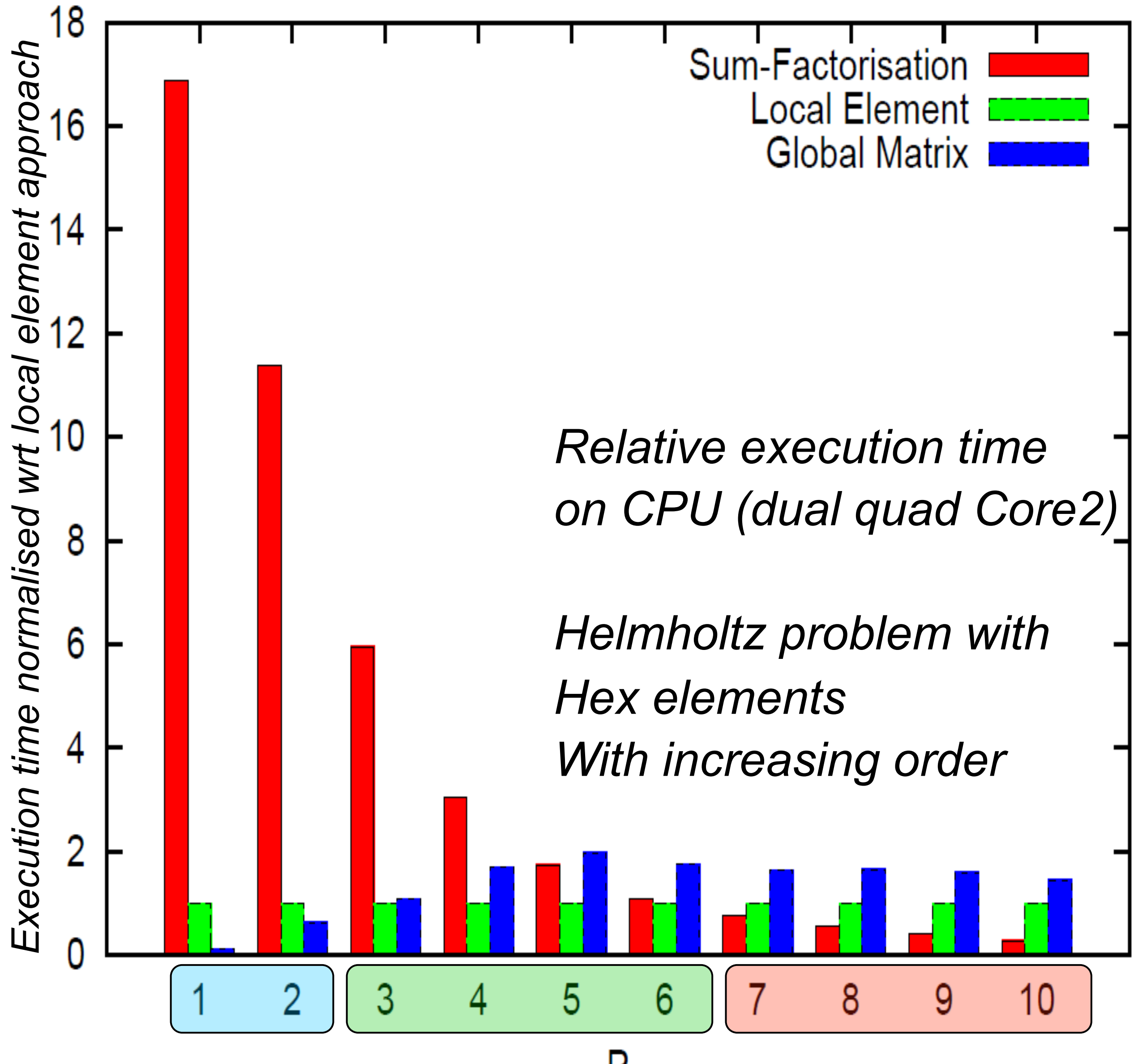
Major savings are possible through aggressive large-scale factorisation

- *#FLOPs for local assembly of pre-multiplied mass matrices of varying complexity over a two-dimensional triangular cell*
- *Forms use an order q Lagrangian basis multiplied with n_f functions of order p, also discretised using a Lagrangian basis.*
- *Columns Q, T and E show #FLOPs for quadrature, tensor contraction and our optimised implementations, respectively*
- *B/E denotes improvement over min(Q,T)*

(Preliminary results presented at FEniCS'11, paper under review)

Mapping the design space – h/p

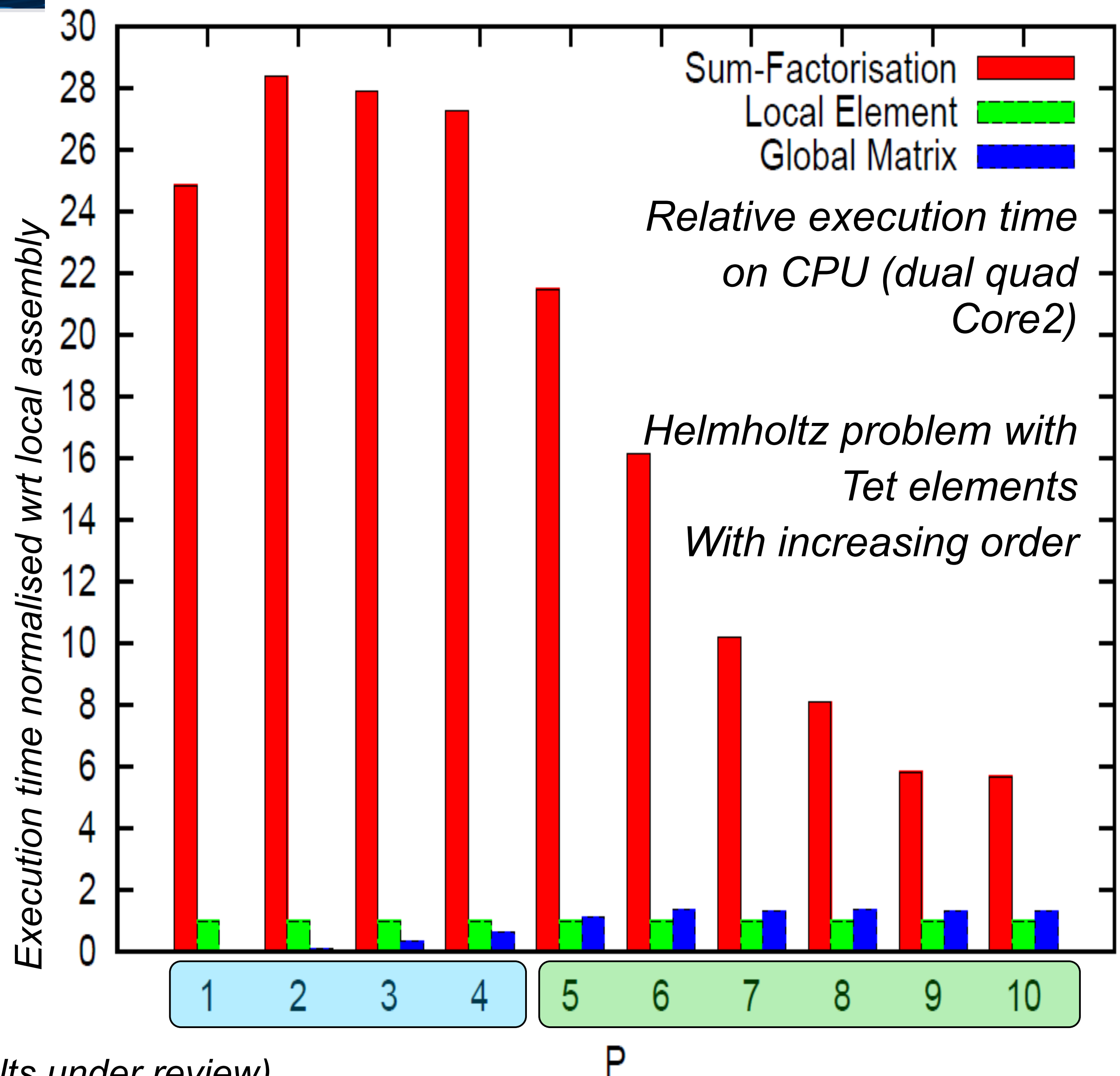
- The balance between local- vs global-assembly depends on multiple factors
- Eg tetrahedral vs hexahedral
- Eg higher-order elements
- Local vs Global assembly is not the only interesting option



(C.D.Cantwell, S.J.Sherwin, R.M.Kirby, P.H.J.Kelly, From h to p efficiently)

Mapping the design space – h/p

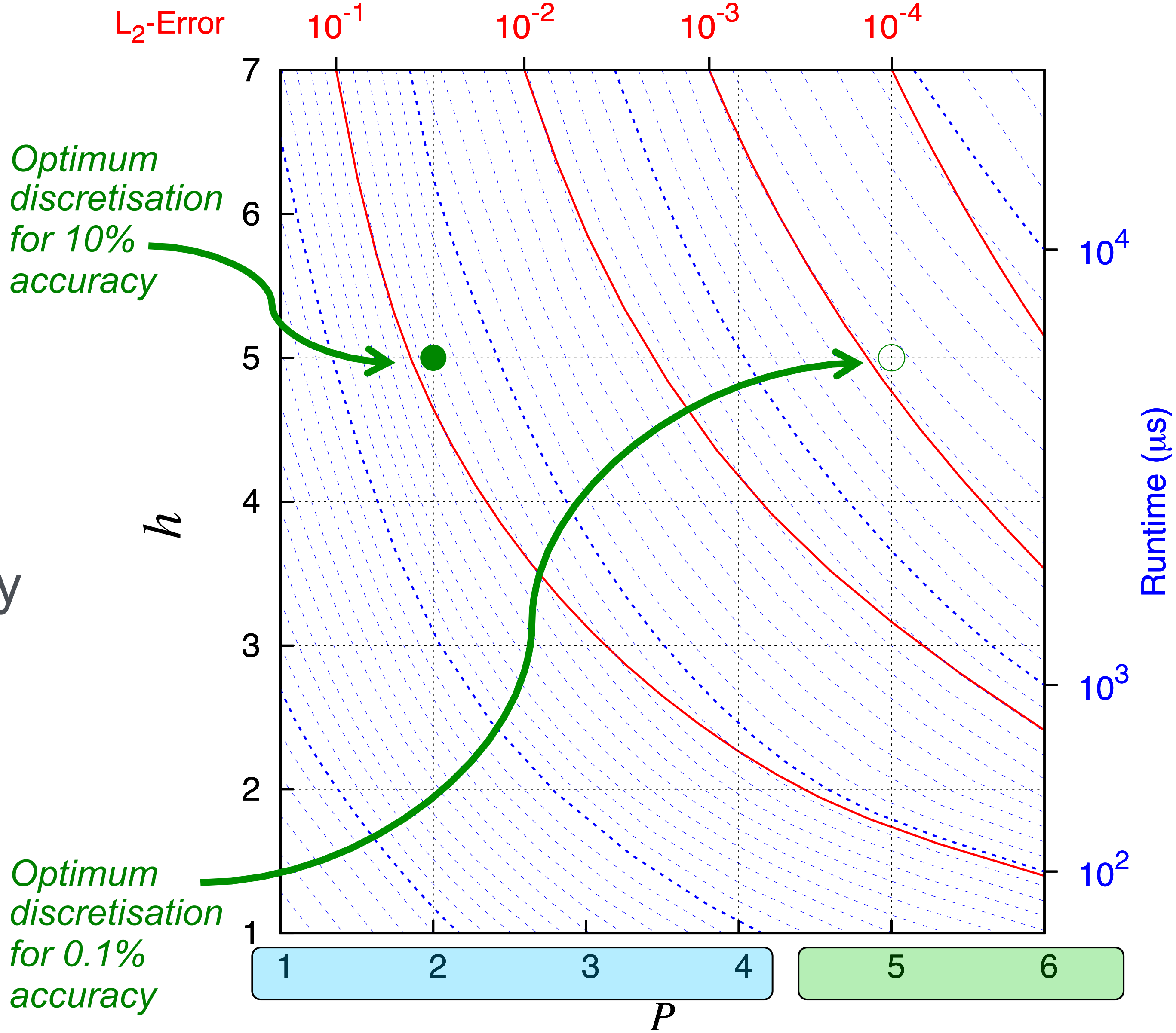
- Contrast: with tetrahedral elements
- Local is faster than global only for much higher-order
- Sum factorisation never wins



(Cantwell et al, provisional results under review)

End-to-end accuracy drives algorithm selection

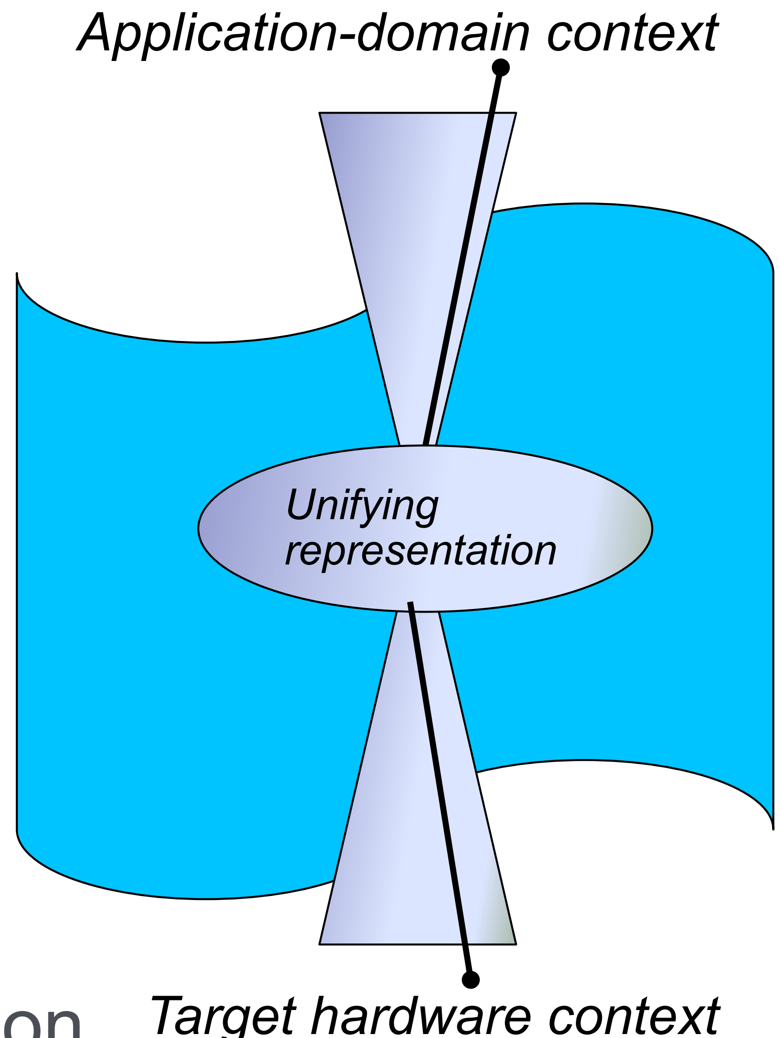
- Helmholtz problem using tetrahedral elements
- What is the best combination of h and p?
- Depends on the solution accuracy required
- Which, in turn determines whether to choose local vs global assembly



Blue dotted lines show runtime of optimal strategy; Red solid lines show L_2 error

Conclusions and Further Work

- From these experiments:
  - Algorithm choice makes a big difference in performance
  - The best choice varies with the target hardware
  - The best choice also varies with problem characteristics and accuracy objectives
- We need to automate code generation
- So we can navigate the design space freely
- And pick the best implementation strategy for each context



Imperial College London

Where this is going

- For OP2:
  - For aeroengine turbomachinery CFD, funded by Rolls Royce and the TSB (the SILOET programme)
  - In progress:
    - For Fluidity, and thus into the Imperial College Ocean Model
  - Feasibility studies being pursued: UK Met Office (“Gung Ho” project), Deutsche Wetterdienst ICON model, Nektar++
- For UFL and our Multicore Form Compiler
  - For Fluidity, supporting automatic generation of adjoint models
- Beyond:
  - Similar DSL ideas for the ONETEP quantum chemistry code
  - Similar DSL ideas for 3D scene understanding

Acknowledgements

- Partly funded by
  - NERC Doctoral Training Grant (NE/G523512/1)
  - EPSRC “MAPDES” project (EP/I00677X/1)
  - EPSRC “PSL” project (EP/I006761/1)
  - Rolls Royce and the TSB through the SILOET programme
  - AMD, Codeplay, Maxeler Technologies