Software abstractions for many-core software engineering

Paul H J Kelly
Group Leader, Software Performance Optimisation
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, Graham Markall, Tiziano Santoro, George Rokos (Software Perf Opt Group, Imperial)
Spencer Sherwin (Aeronautics, Imperial), Chris Cantwell (Cardio-mathematics group, Mathematics, Imperial)

Imperial College London

What we are doing....

Rolls-Royce HYDRA turbomachinery CFD

LAMMPS – granular flow

FENiCS finite-element PDE generator

Fluidity and the Imperial College Ocean Model (ICOM)

Multicore Form Compiler

Finite-volume CFD

Particle problems – molecular dynamics

Finite-element assembly

Mesh adaptation

Moving meshe

Mixed meshes

P-adaptivity

Pair_gen for LAMMPS

OP2: parallel loops over unstructured meshes

OP2.1: extended with dynamic meshes

OP2.2: extended with sparse matrices

OP2.3: with fully-abstract graphs

OP2.4: mixed and piecewise structured meshes

Fortran & C/C++ OP2 compiler

CUDA/ OpenCL

OpenMP

MPI

SSE/AVX

FPGAs

?

- Roadmap: applications drive DSLs, delivering performance portability

The message

Imperial College London

- Three slogans
  - Generative, instead of transformative optimisation
  - Get the abstraction right, to isolate numerical methods from mapping to hardware
  - Build vertically, learn horizontally
- Three stories
  - Domain-specific active library examples
  - General framework: access-execute descriptors
  - The value of generative and DSL techniques

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?

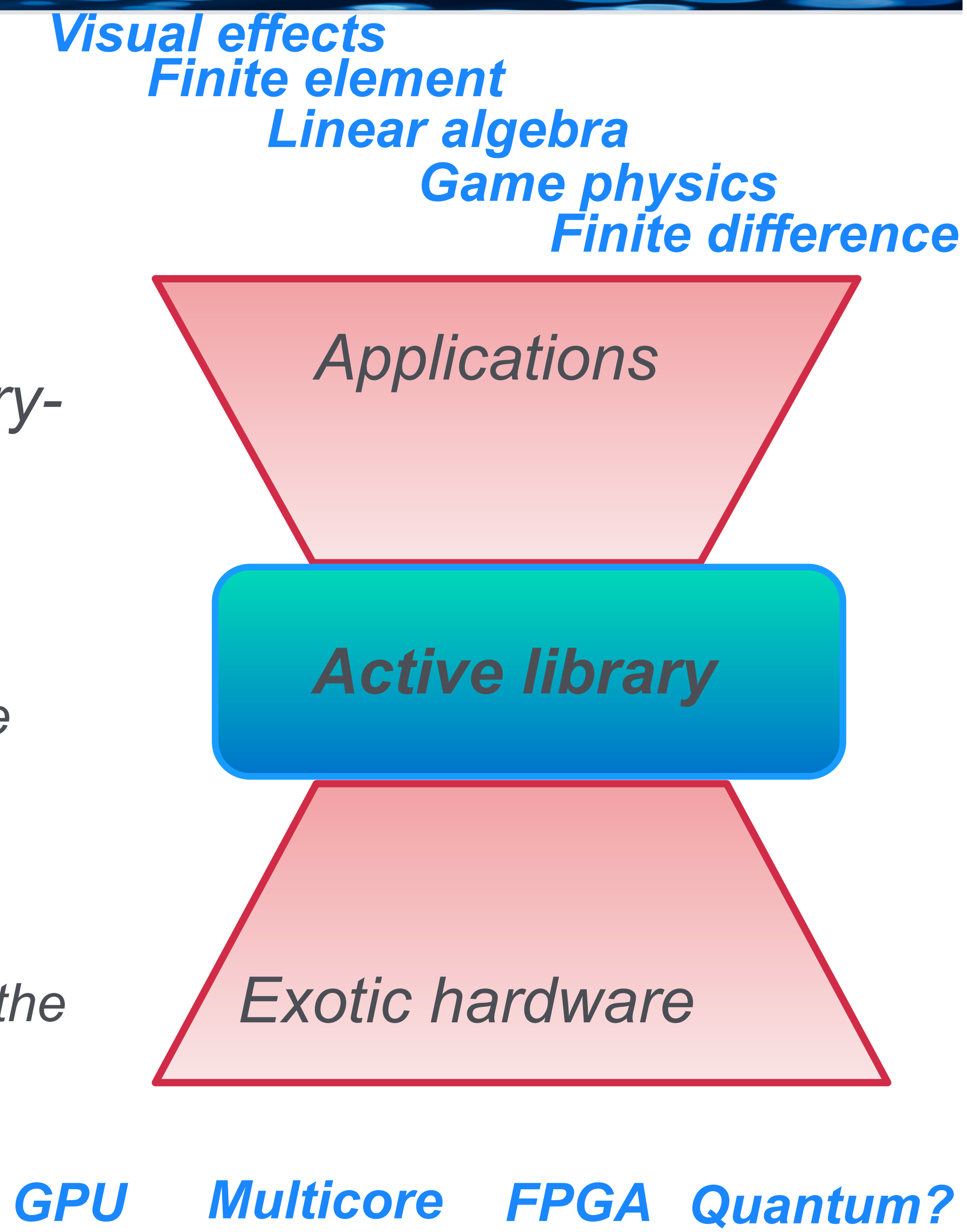
It's a software engineering problem

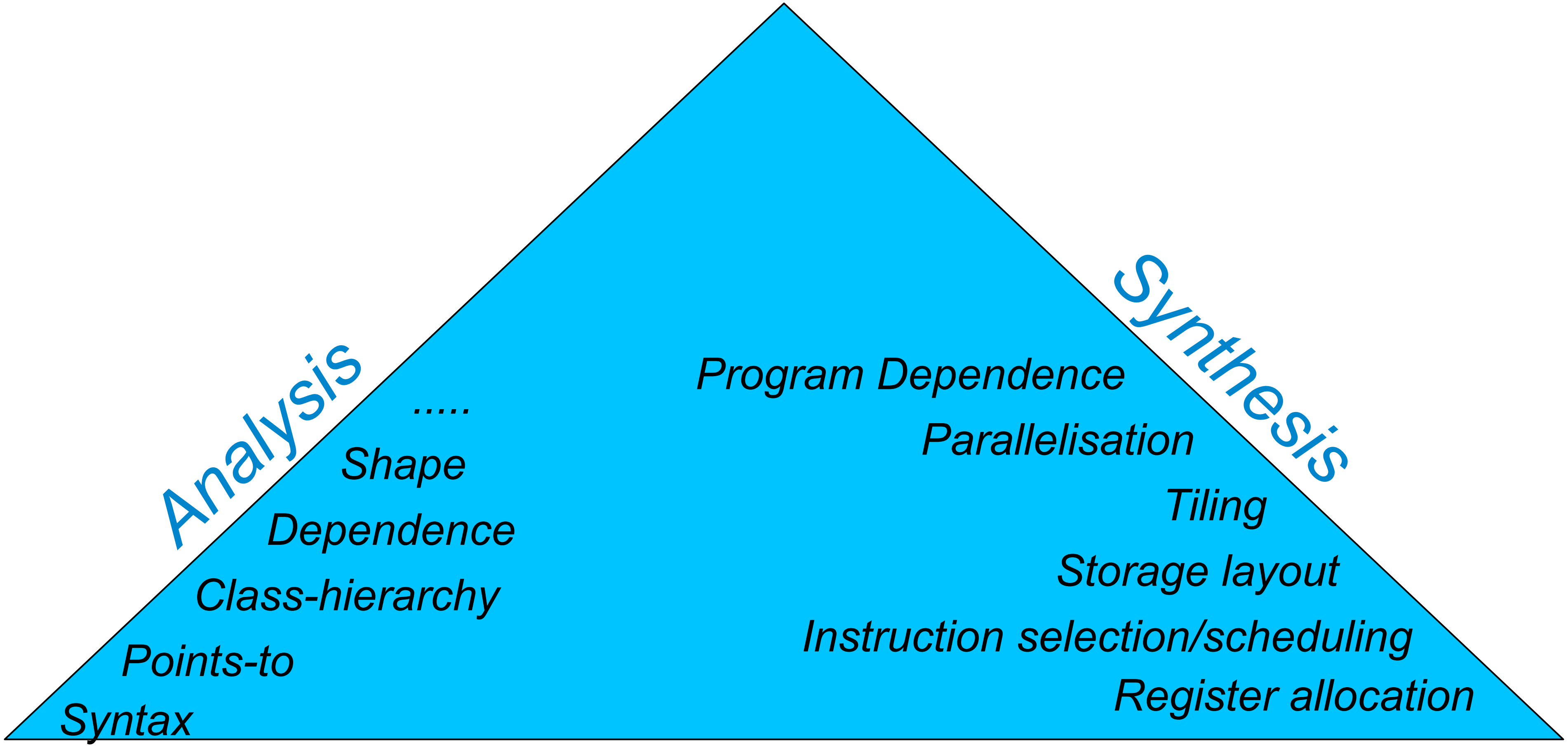
Imperial College London

Active libraries and DSLs

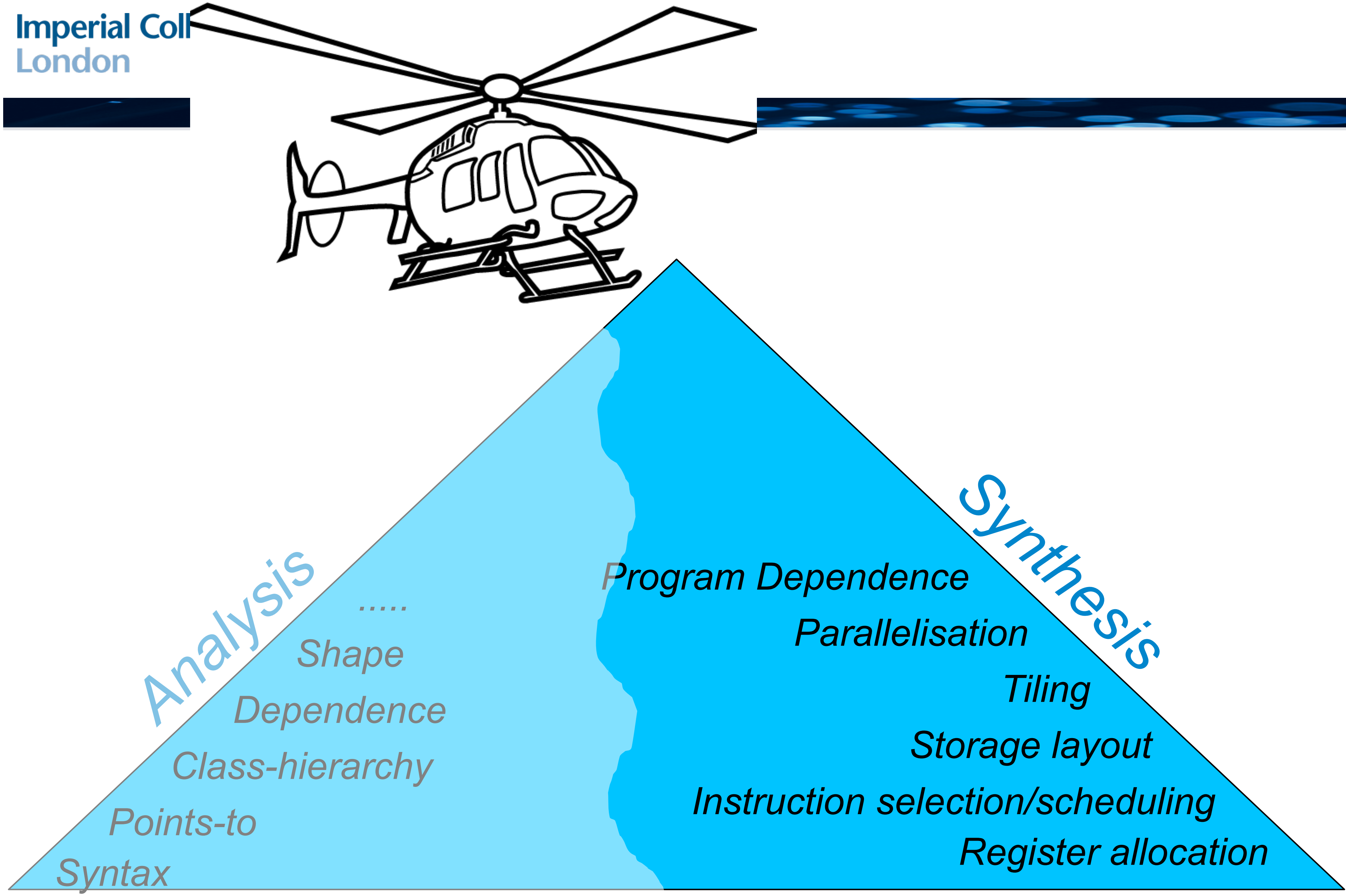
- *Domain-specific languages...*
- *Embedded DSLs*
- *Active libraries*
  - *Libraries that come with a mechanism to deliver library-specific optimisations*

- *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*

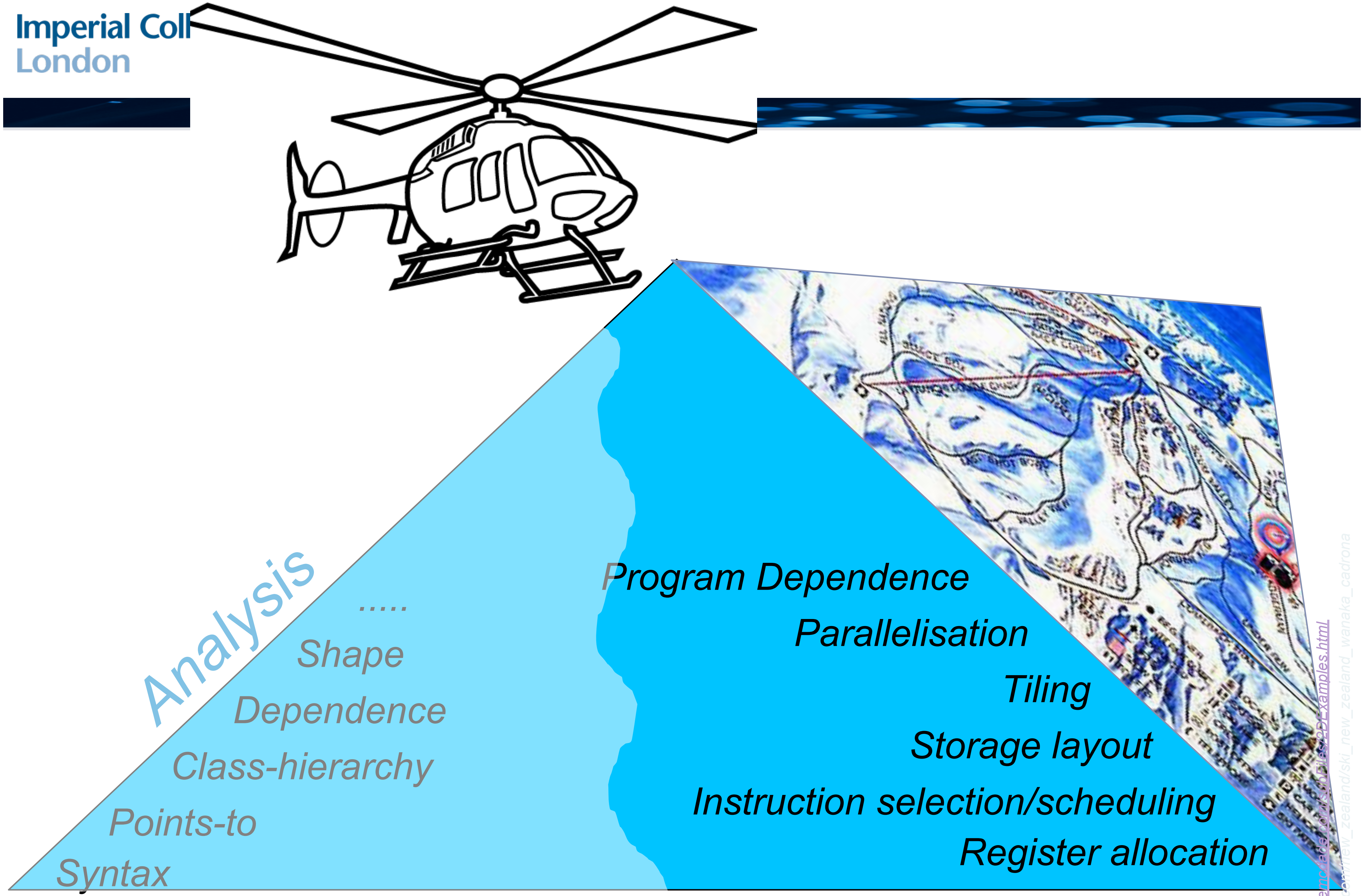




- Classical compilers have two halves



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



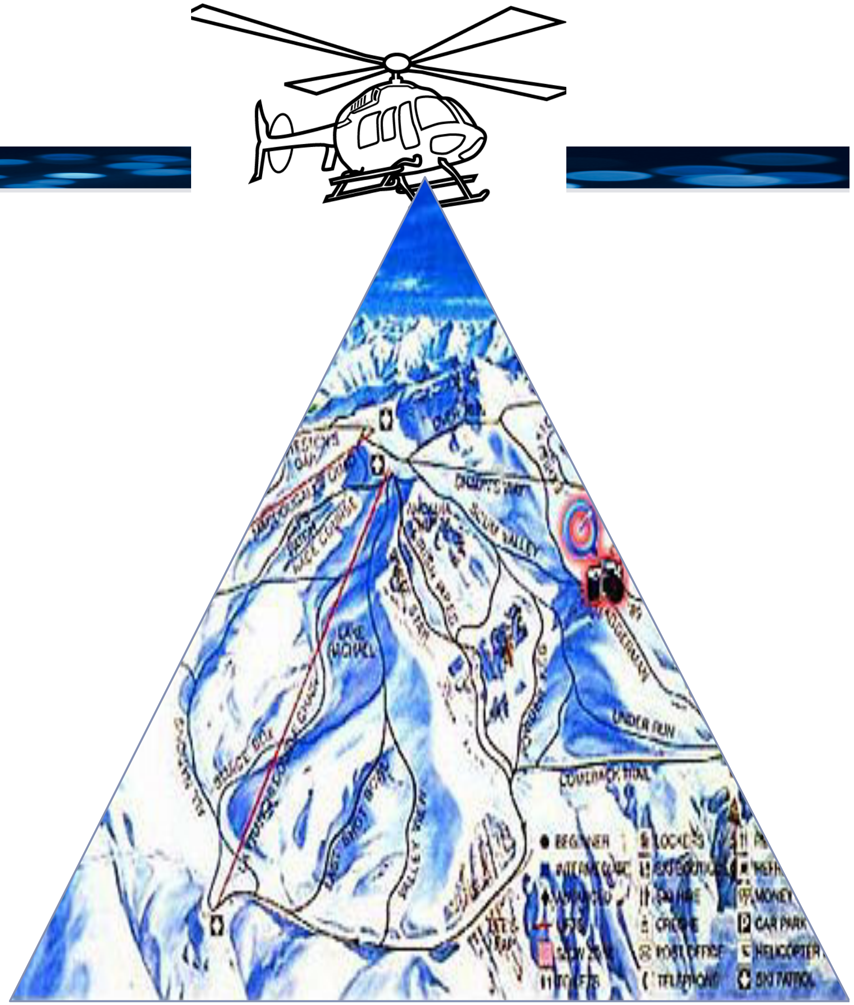
http://www.nikkiemcdade.com/subFiles/2DExamples.html

http://www.ginz.com/new_zealand/ski_new_zealand_wanaka_cadrona

- 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

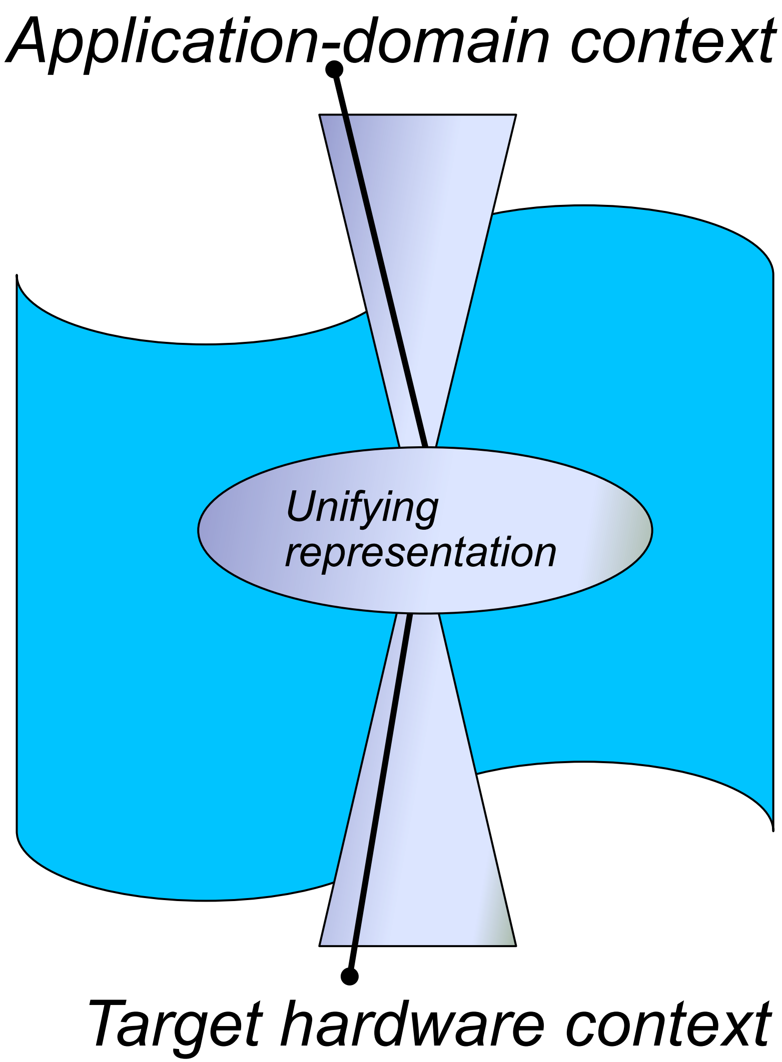
Capture dependence and communication in programs over richer data structures

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

Imperial College London

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

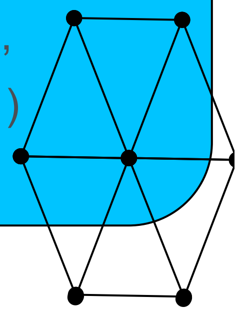


OP2 – a decoupled access-execute active library for unstructured mesh computations

```
// 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 );
```

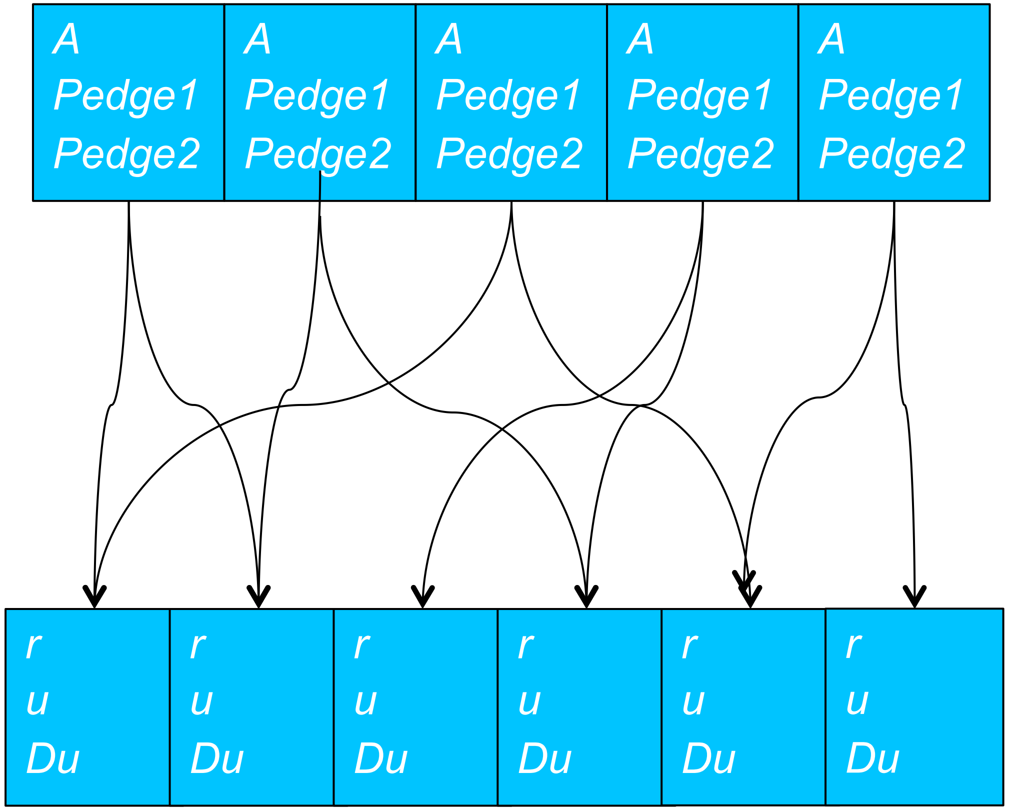
Example – Jacobi solver

```
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 )
     );
}
```

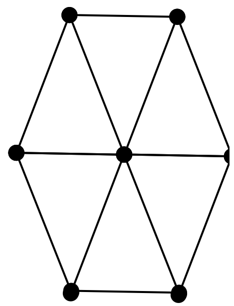


OP2- Data model

```
// 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]);
}
```

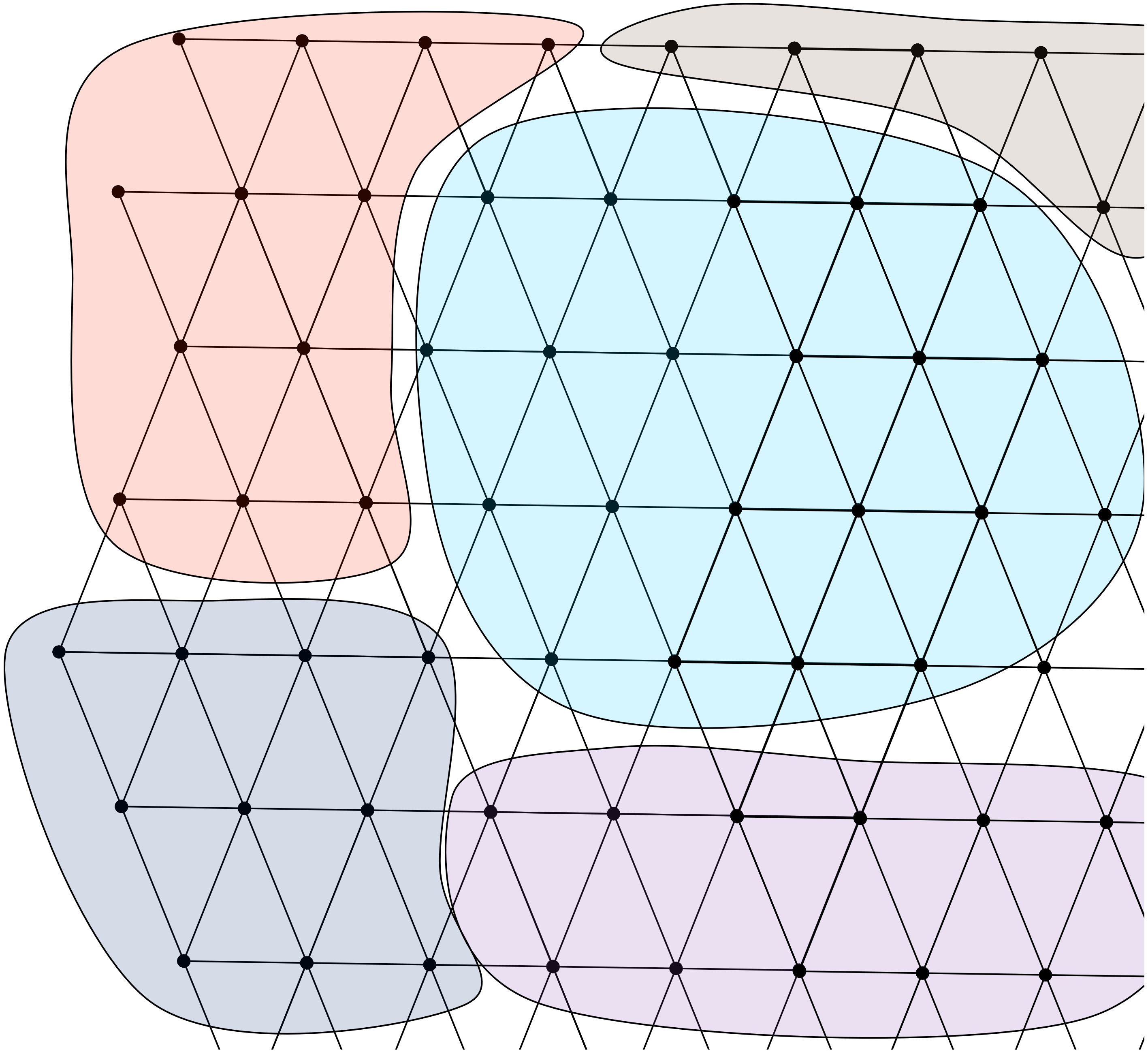
```
float u_sum, u_max, beta = 1.0f;

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"

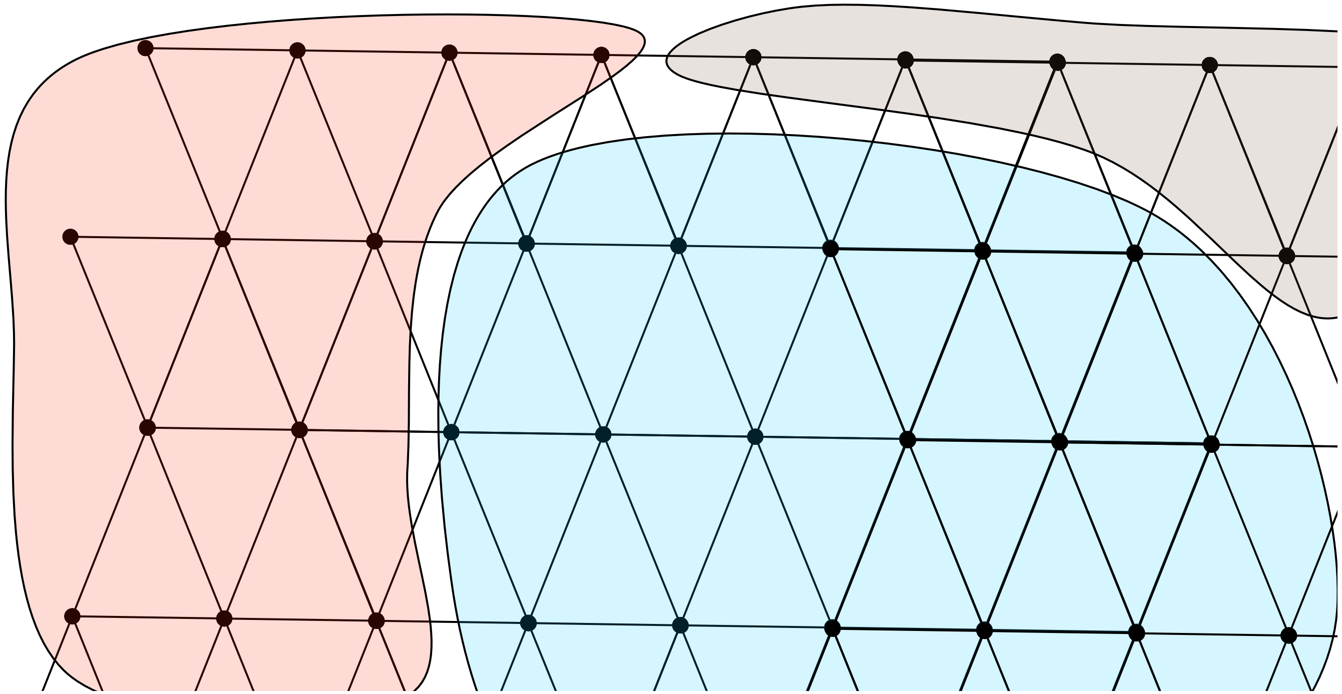
Example – Jacobi solver

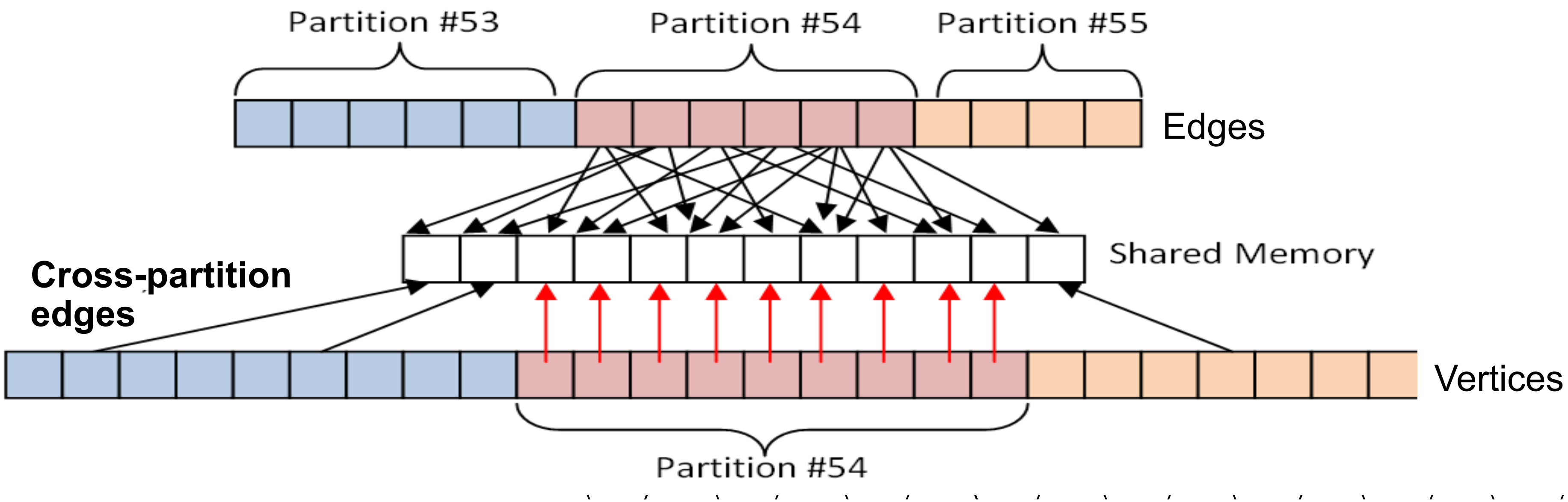
- Two key optimisations:
- Partitioning
- Colouring

- *Here we focus on GPU and multicore implementation*
- *We also have MPI-level parallelisation*
- *Exploring SSE/AVX*
- *And FPGA*

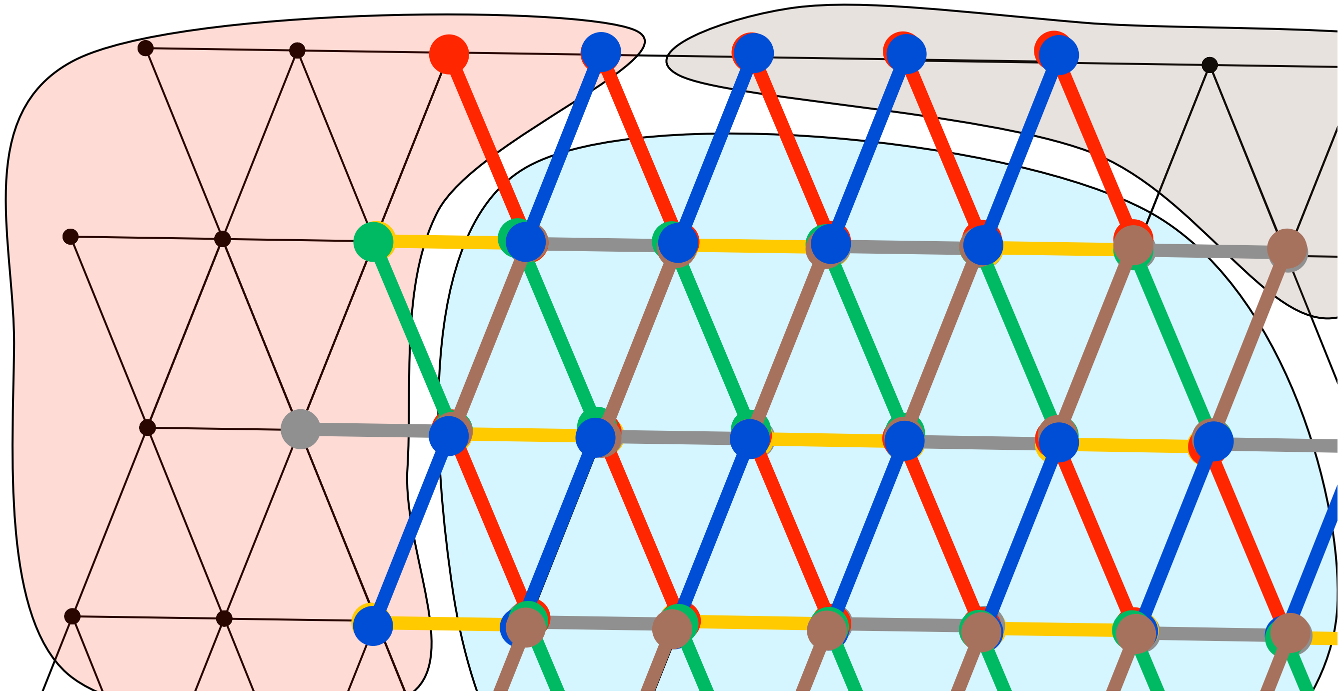


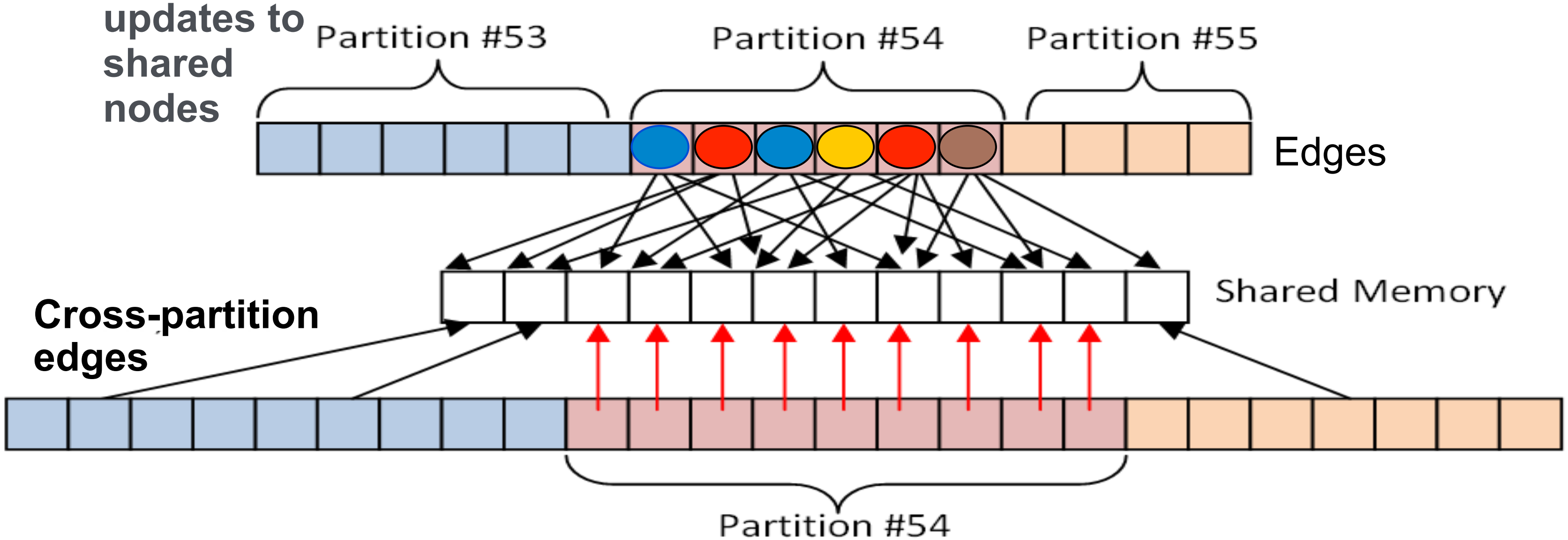
- Two key optimisations:
- **Partitioning**
- Colouring



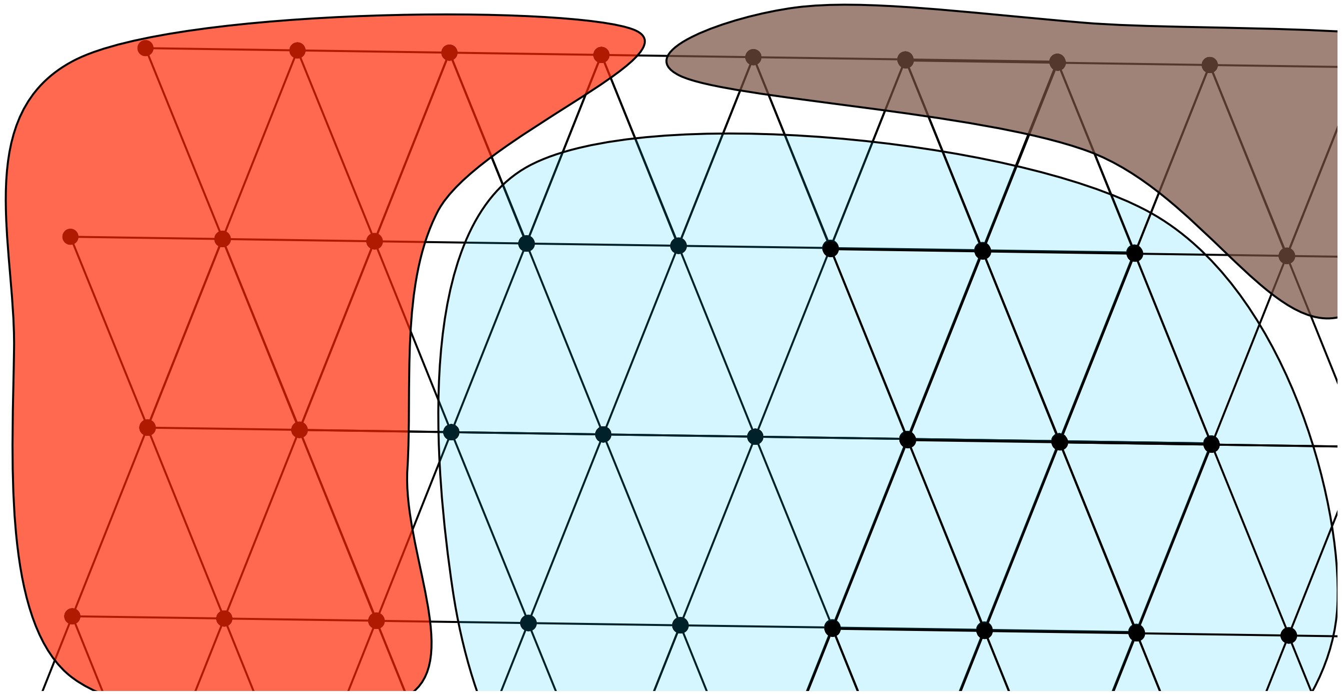


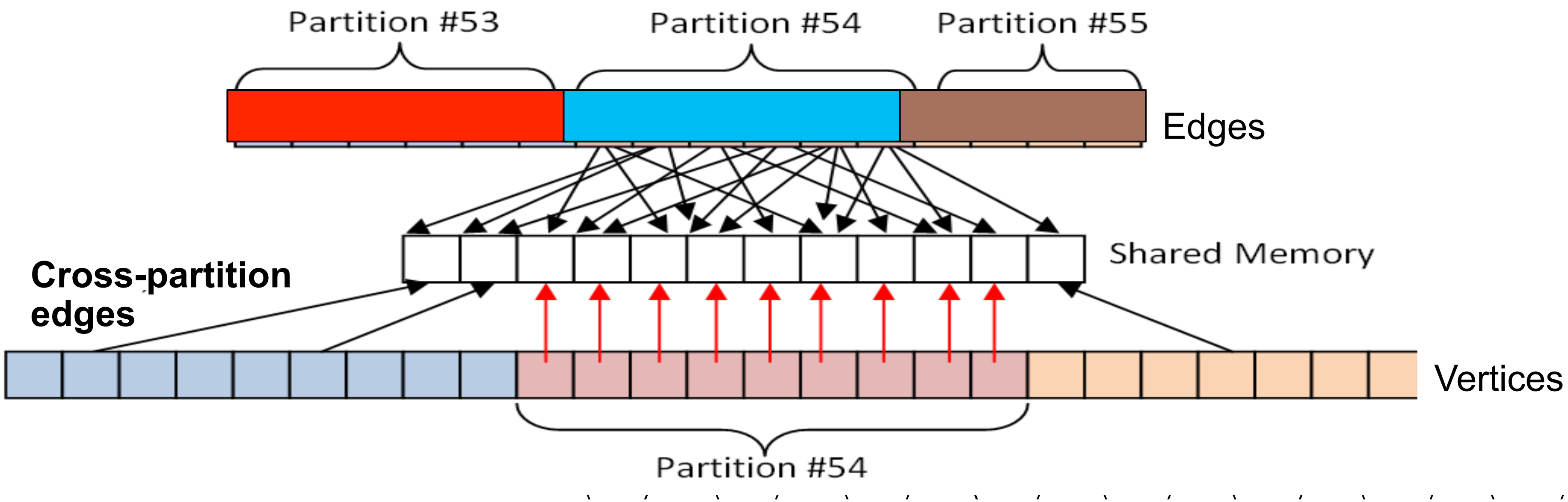
- Two key optimisations:
- Partitioning
- **Colouring**
  - **Elements of the edge set are coloured to avoid races due to concurrent updates to shared nodes**



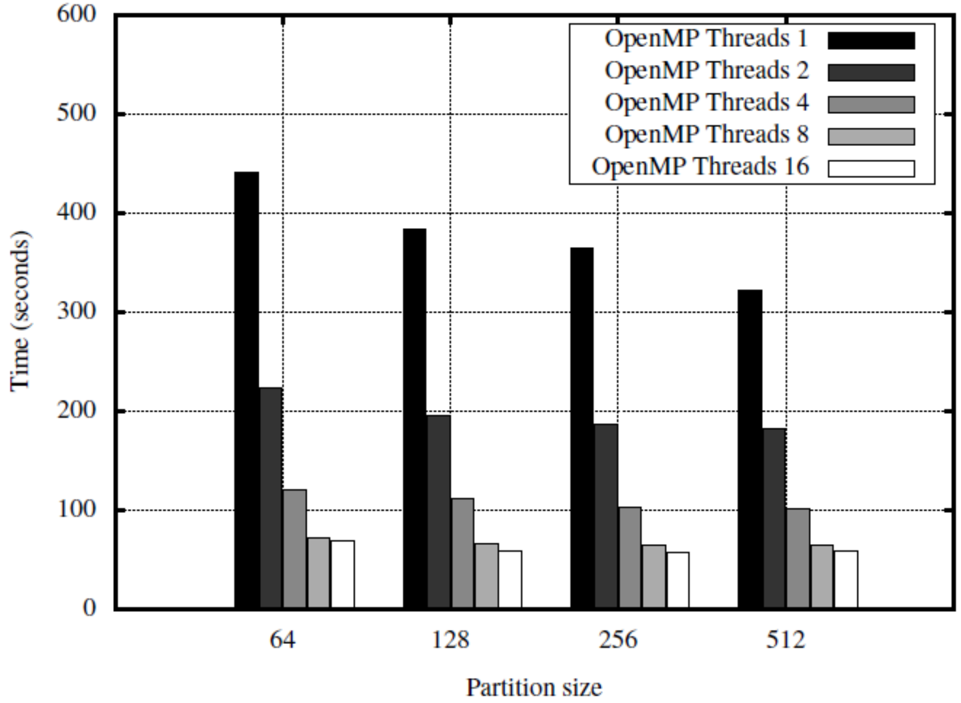


- Two key optimisations:
- **Partitioning**
- Colouring
  - At two levels

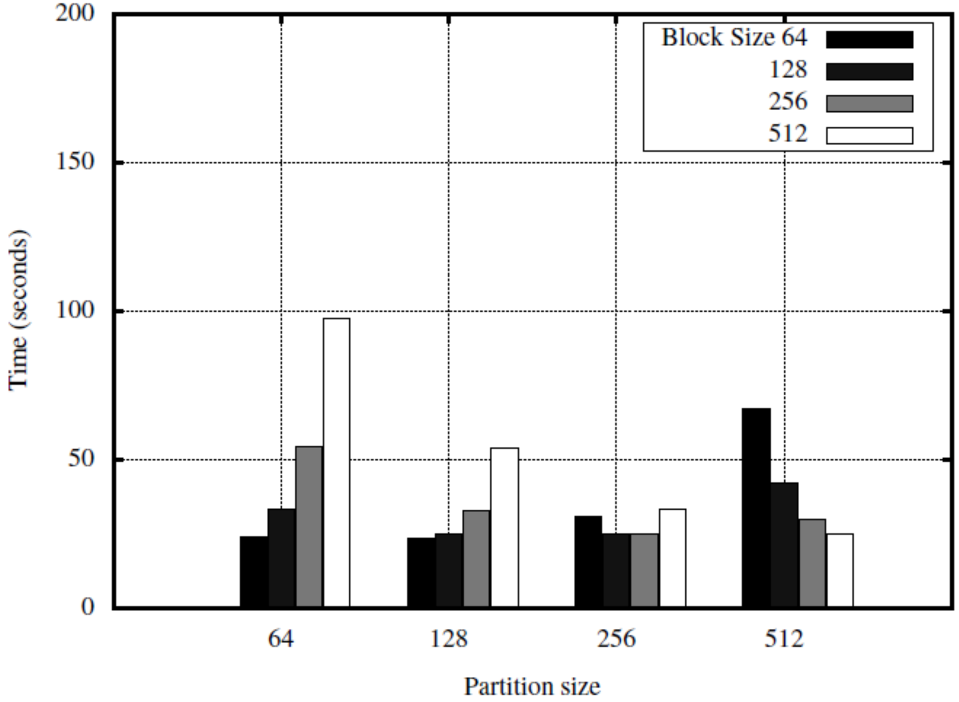




OP2 - performance

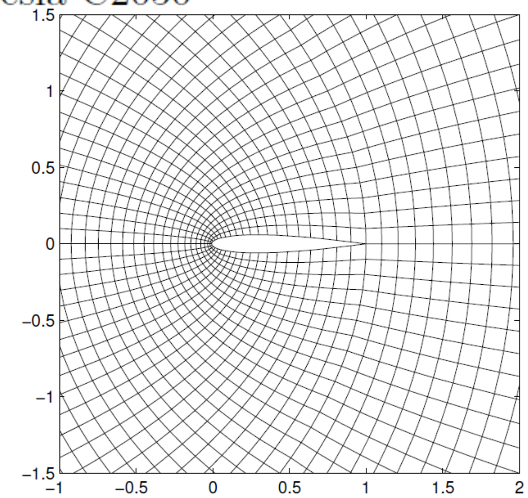


(a) Intel Xeon E5540 (Nehalem) (ICC 11.1)



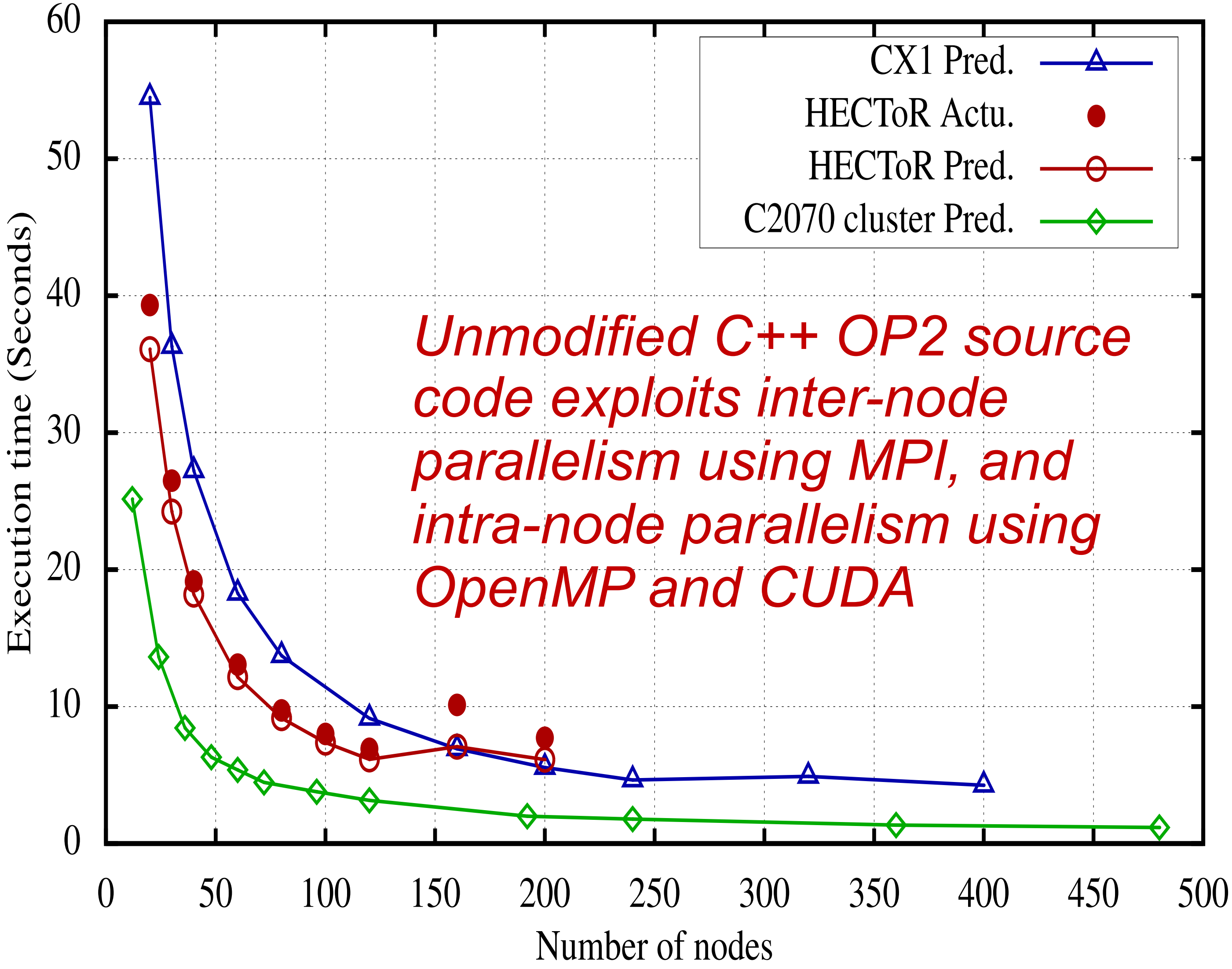
(b) Tesla C2050

- 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



Combining MPI, OpenMP and CUDA

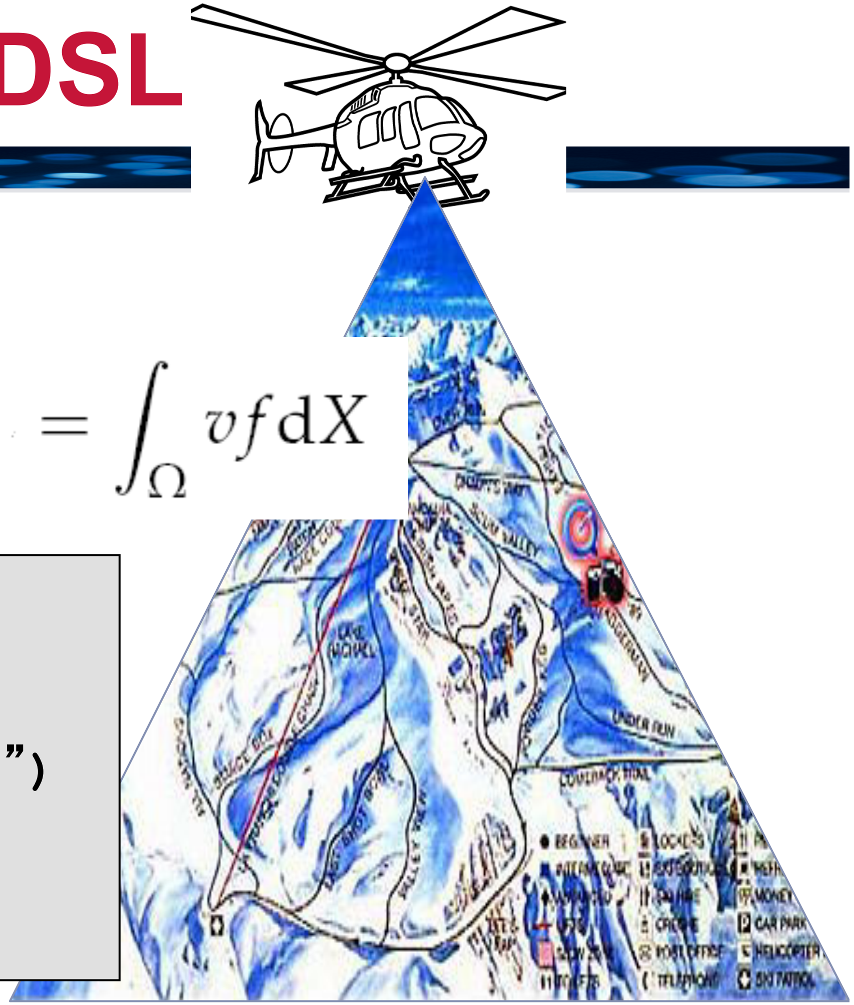
- *non-linear 2D inviscid airfoil code*
- *26M-edge unstructured mesh*
- *1000 iterations*
- *Analytical model validated on up to 120 Westmere X5650 cores and 1920 HECToR (Cray XE6) cores*



(Preliminary results under review)

Imperial College London

A higher-level DSL



- Solving: $\nabla^2 u = f$
- Weak form: $\int_\Omega \nabla v \cdot \nabla u \mathrm{d}X = \int_\Omega v f \mathrm{d}X$

  (Ignoring boundaries)

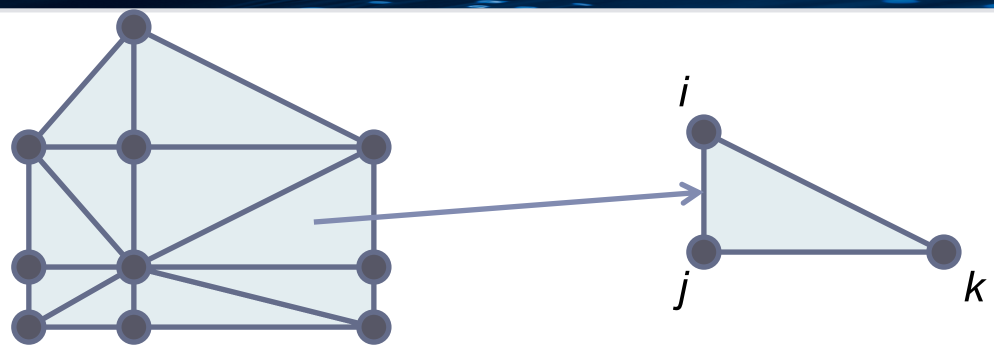
```
Psi = state.scalar_fields("psi")
v=TestFunction(Psi)
u=TrialFunction(Psi)
f=Function(Psi, "sin(x[0])+cos(x[1])")
A=dot(grad(v),grad(u))*dx
RHS=v*f*dx
Solve(Psi,A,RHS)
```

UFL – Unified Form Language
(FEniCS project, http://fenicsproject.org/):
A domain-specific language for generating finite element discretisations of variational forms

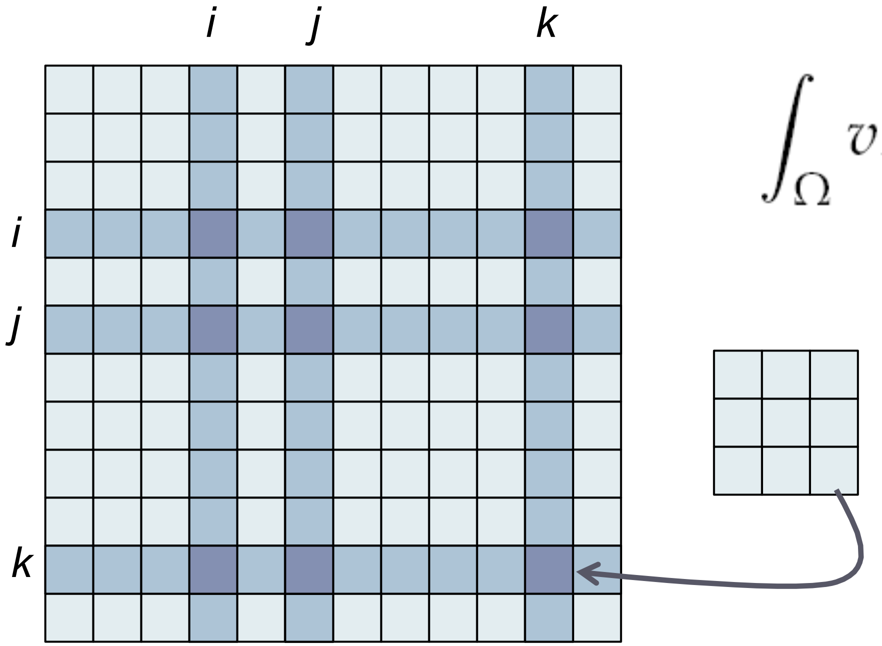
Specify application requirements, leaving implementation to select radically-different solution approaches

Imperial College London

The FE Method: computation overview

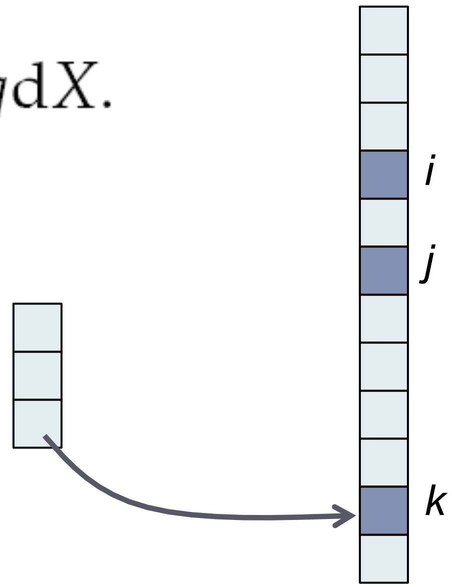



```
do element = 1,N
  assemble(element)
end do
```



$$\int_\Omega vL(u^\delta)\mathrm{d}X = \int_\Omega vq\mathrm{d}X.$$

$$Ax = b$$



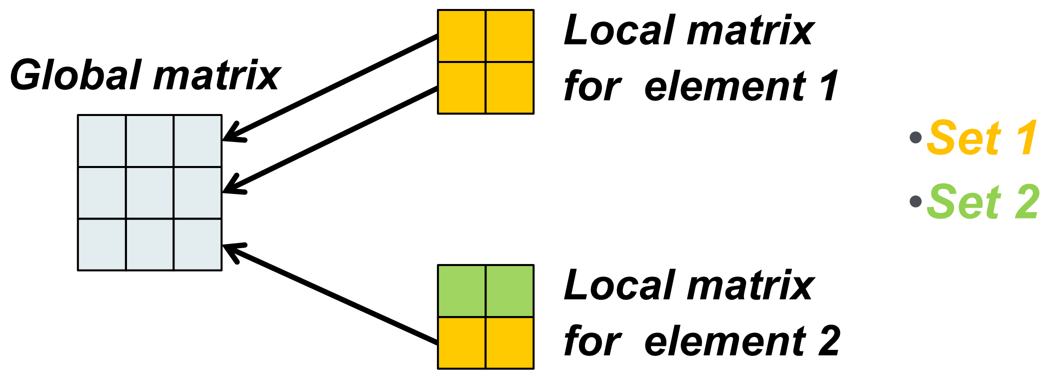
- Key data structures: Mesh, dense local assembly matrices, sparse global system matrix, and RHS vector

Global Assembly – GPU Issues

Parallelising the global assembly leads to performance/ correctness issues:

- Bisection search: uncoalesced accesses, warp divergence
- Contending writes: atomic operations, colouring

$$\mathcal{A} = \begin{bmatrix} 1_0 & & \\ & 1_0 & \\ & 1_1 & \\ & & 1_0 \end{bmatrix}$$



- In some circumstances we can avoid building the global system matrix altogether
- Goal: get the UFL compiler to pick the best option

The Local Matrix Approach

- Why do we assemble *M*?

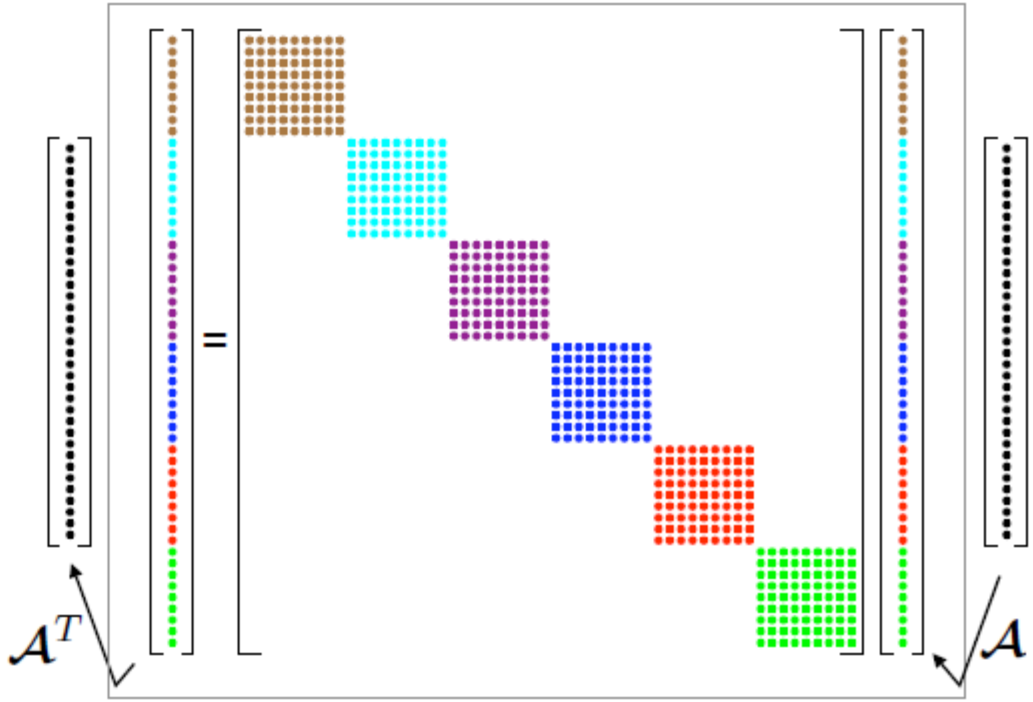
We need to solve $y = Mv$ *where* $M = A^T M^e A$

- In the *Local Matrix Approach* we recompute this, instead of storing it:

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

- *b* is explicitly required
- Assemble it with an SpMV:

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



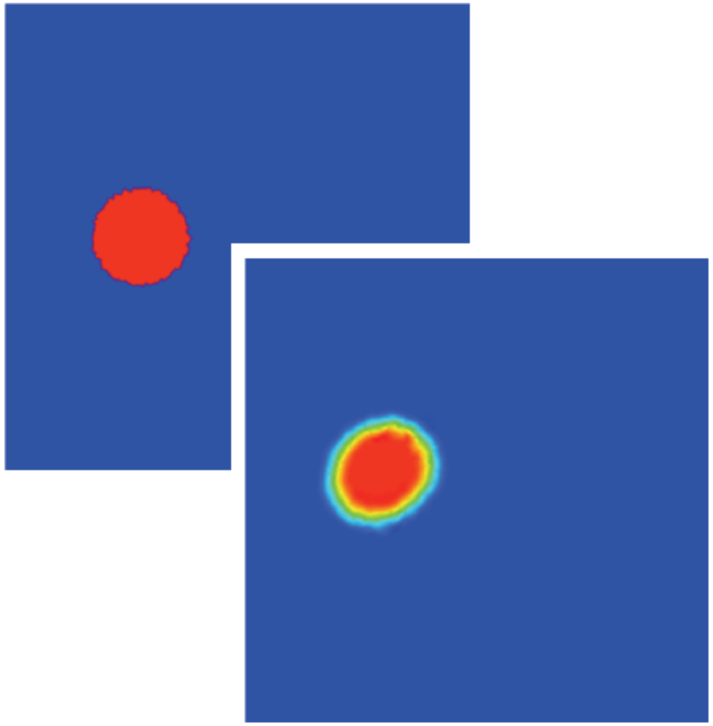
Imperial College London

Test Problem Implementation

- Advection-Diffusion Equation:

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

- Solved using a split scheme:
  - Advection: Explicit RK4
  - Diffusion: Implicit theta scheme
- GPU code: expanded data layouts, with Addto or LMA
- CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010] (Implemented within Fluidity)
- Double Precision arithmetic
- Simulation run for 200 timesteps



- *Simplified CFD test problem*

Imperial College London

Test Platforms

- Nvidia 280GTX:
  - 240 stream processors: 30 multiprocessors with 8 SMs each
  - 1GB RAM (4GB available in Tesla C1060)
- NVidia 480GTX:
  - 480 stream processors: 15 multiprocessors with 32 SMs each
  - 1.5GB RAM (3GB available in Tesla C2050, 6GB in Tesla C2060)
- AMD Radeon 5870:
  - 1600 stream processors: 20 multiprocessors with 16 5-wide SIMD units
  - 1GB RAM (768MB max usable)
- Intel Xeon E5620:
  - 4 cores
  - 12GB RAM

Software:

Ubuntu 10.04

Intel Compiler 10.1 for Fortran (`-O3` flag)

NVIDIA CUDA SDK 3.1 for CUDA

ATI Stream SDK 2.2 for OpenCL

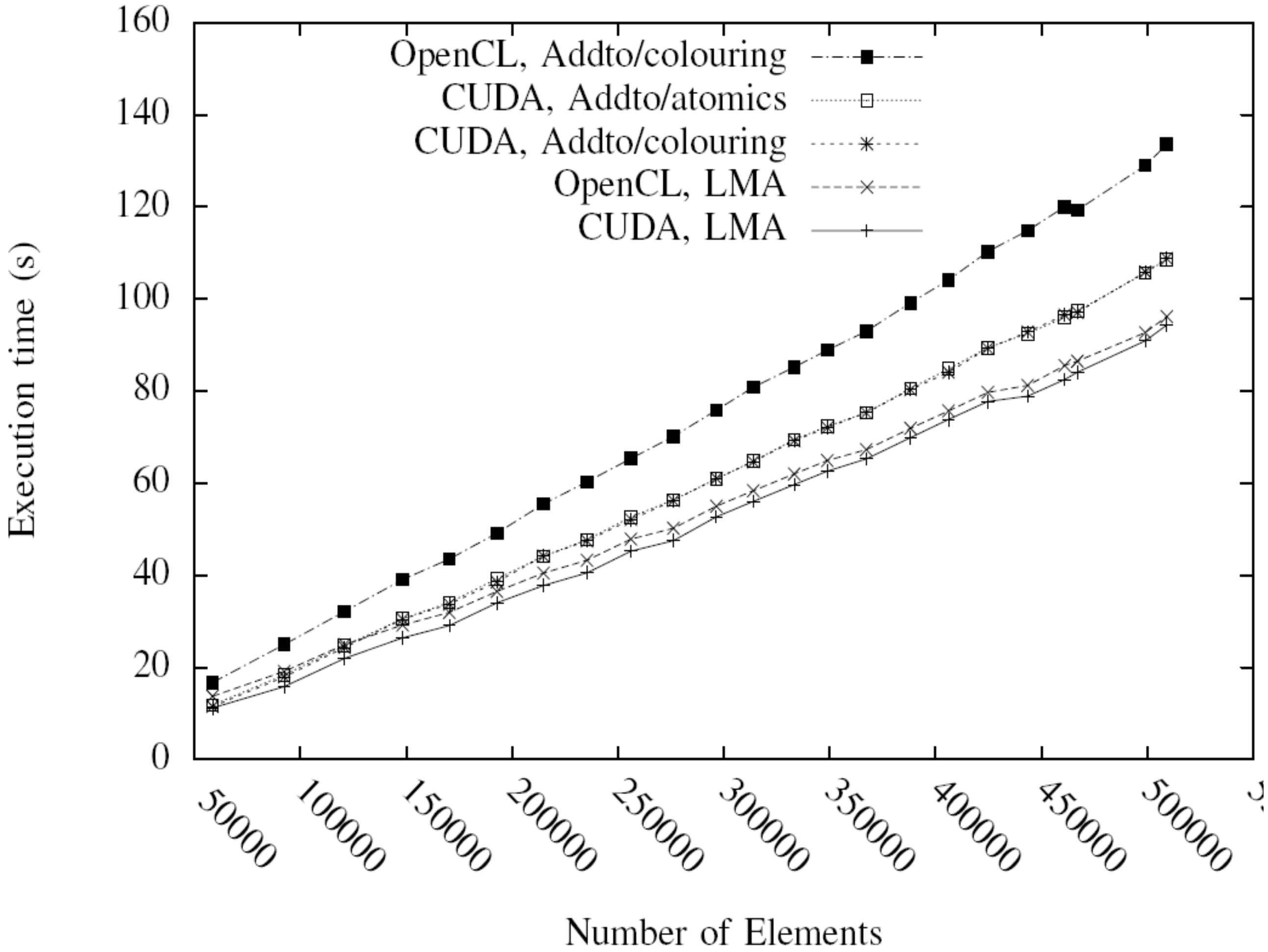
Linear Solver:

CPU: PETSc [Balay et al., 2010]

CUDA Conjugate Gradient Solver [Markall & Kelly, 2009], ported to OpenCL

Fermi Execution times

- Advection-Diffusion Equation:
$$\frac{\partial T}{\partial t} + \boldsymbol{u}\,\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla T$$
- Solved using a split scheme:
  - Advection: Explicit RK4
  - Diffusion: Implicit theta scheme
- GPU code: expanded data layouts, with Addto or LMA
- CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010] (Implemented within Fluidity)
- Double Precision arithmetic
- Simulation run for 200 timesteps



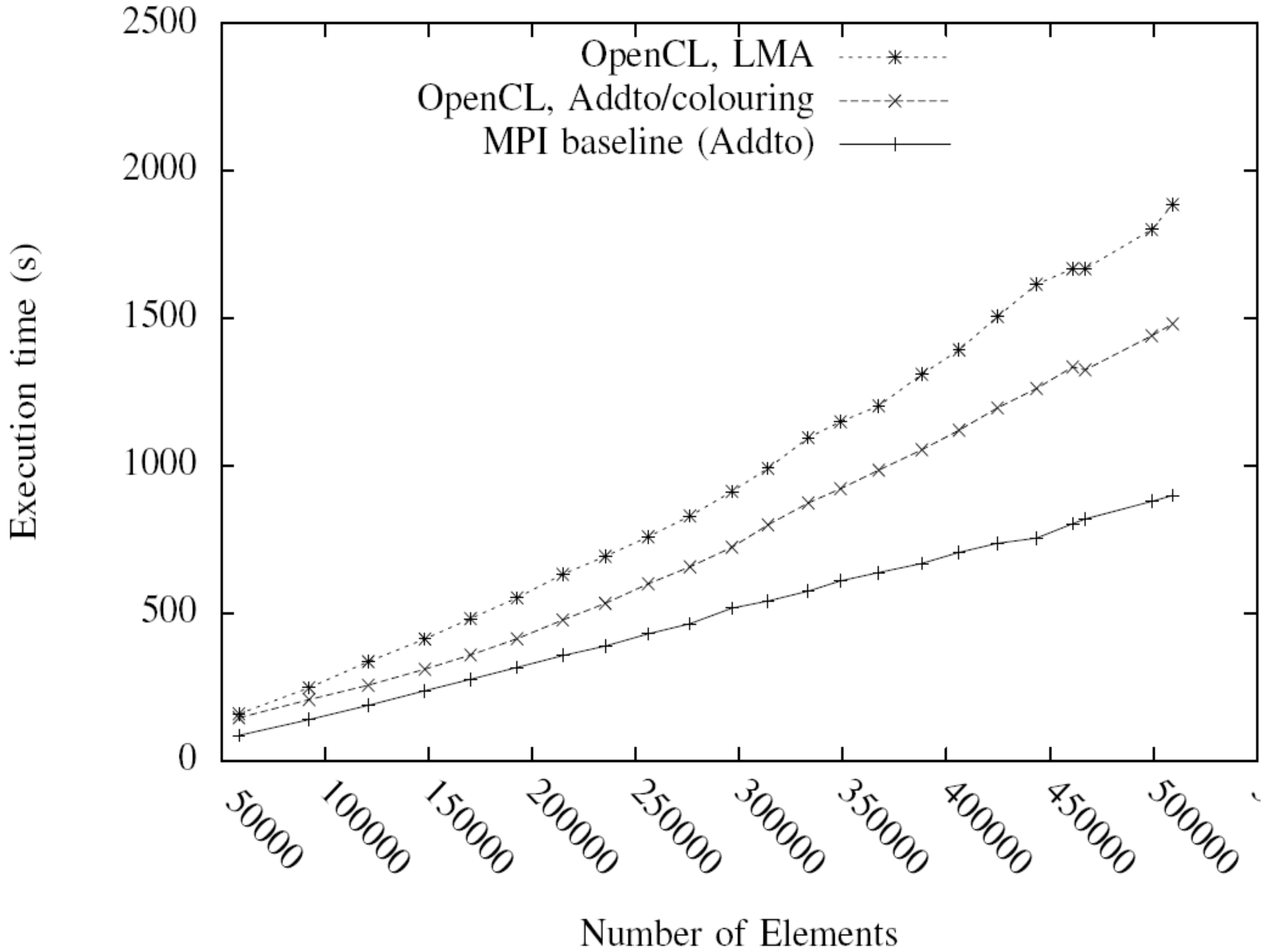
- On the 480GTX ("Fermi") GPU, local assembly is more than 10% slower than the addto algorithm (whether using atomics or with colouring to avoid concurrent updates)

Intel 4-core E5620 (Westmere EP)

- Advection-Diffusion Equation:

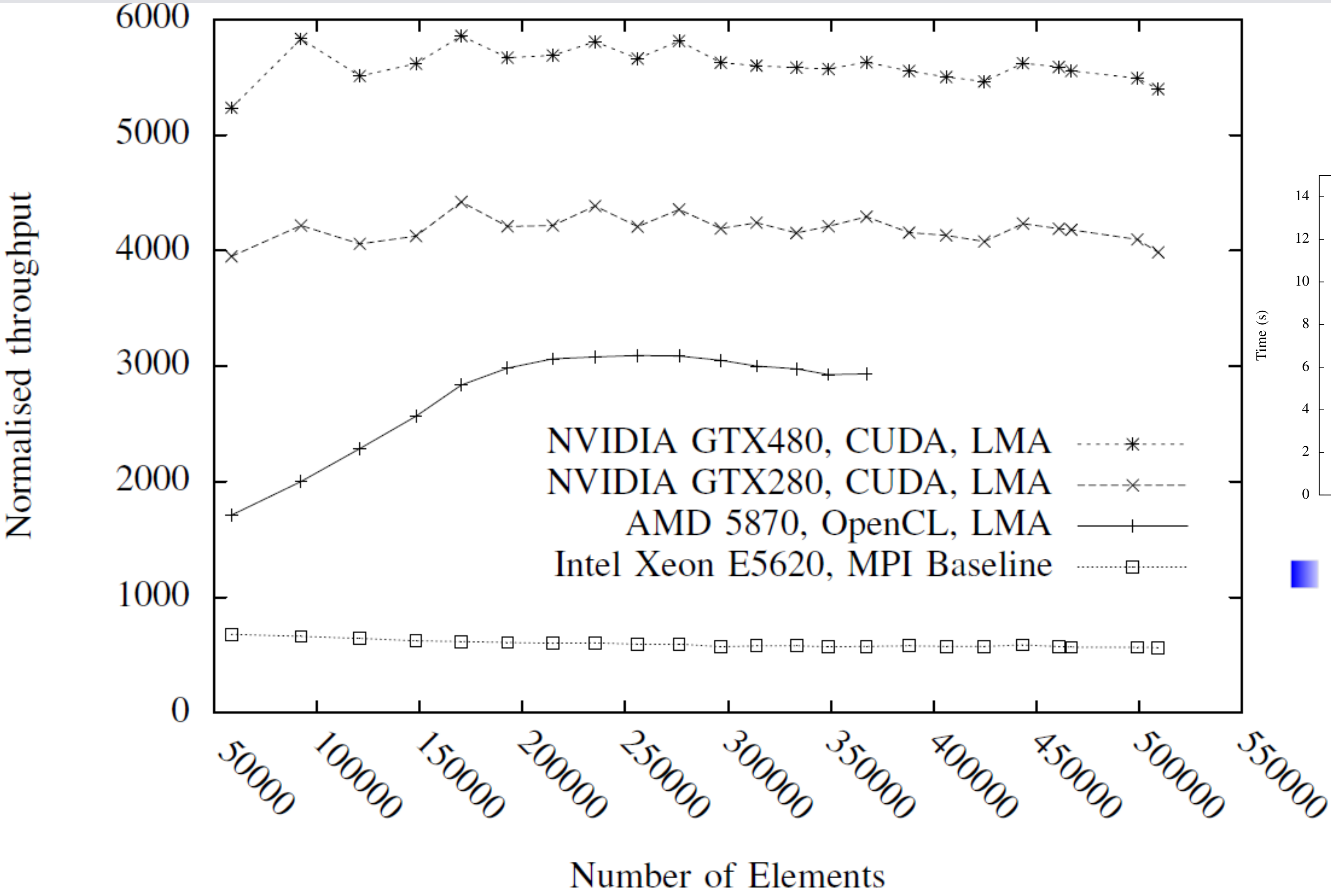
$$\frac{\partial T}{\partial t} + \boldsymbol{u}\,\nabla T = \nabla \cdot \bar{\bar{\mu}} \cdot \nabla \mathrm{T}$$

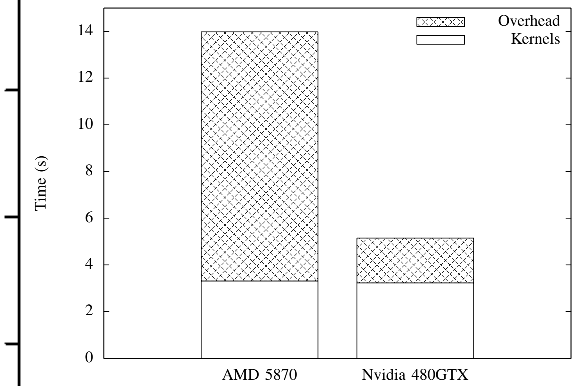
- Solved using a split scheme:
  - Advection: Explicit RK4
  - Diffusion: Implicit theta scheme
- GPU code: expanded data layouts, with Addto or LMA
- CPU baseline code: indirect data layouts, with Addto [Vos et al., 2010] (Implemented within Fluidity)
- Double Precision arithmetic
- Simulation run for 200 timesteps



- On the quad-core Intel Westmere EP system, the local matrix approach is slower. Using Intel's compiler, the baseline code (using addtos and without data expansion) is faster still

Throughput compared to CPU Implementation





- *AMD 5870 and GTX480 kernel times very similar; older AMD drivers incurred overheads*

- Throughput of best GPU implementations relative to CPU (quad-core Westmere E5620)

(preliminary results, esp the AMD numbers)

Imperial College London

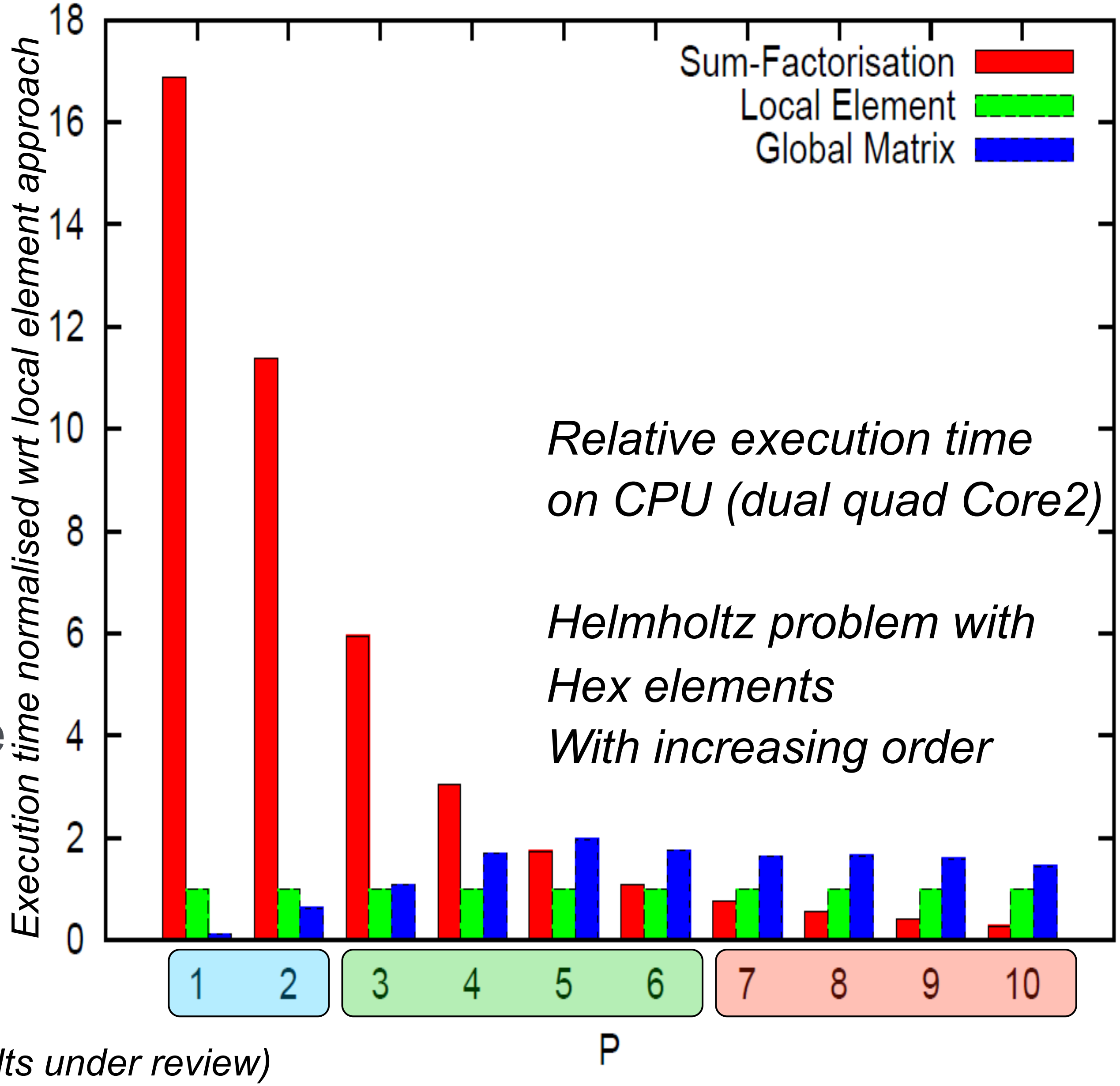
Summary of results

- The Local Matrix Approach is fastest on GPUs
- Global assembly with colouring is fastest on CPUs

- Expanded data layouts allow coalescing and higher performance on GPUs
- Accessing nodal data through indirection is better on CPU due to cache, lower memory bandwidth, and arithmetic throughput

Mapping the design space – h/p

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

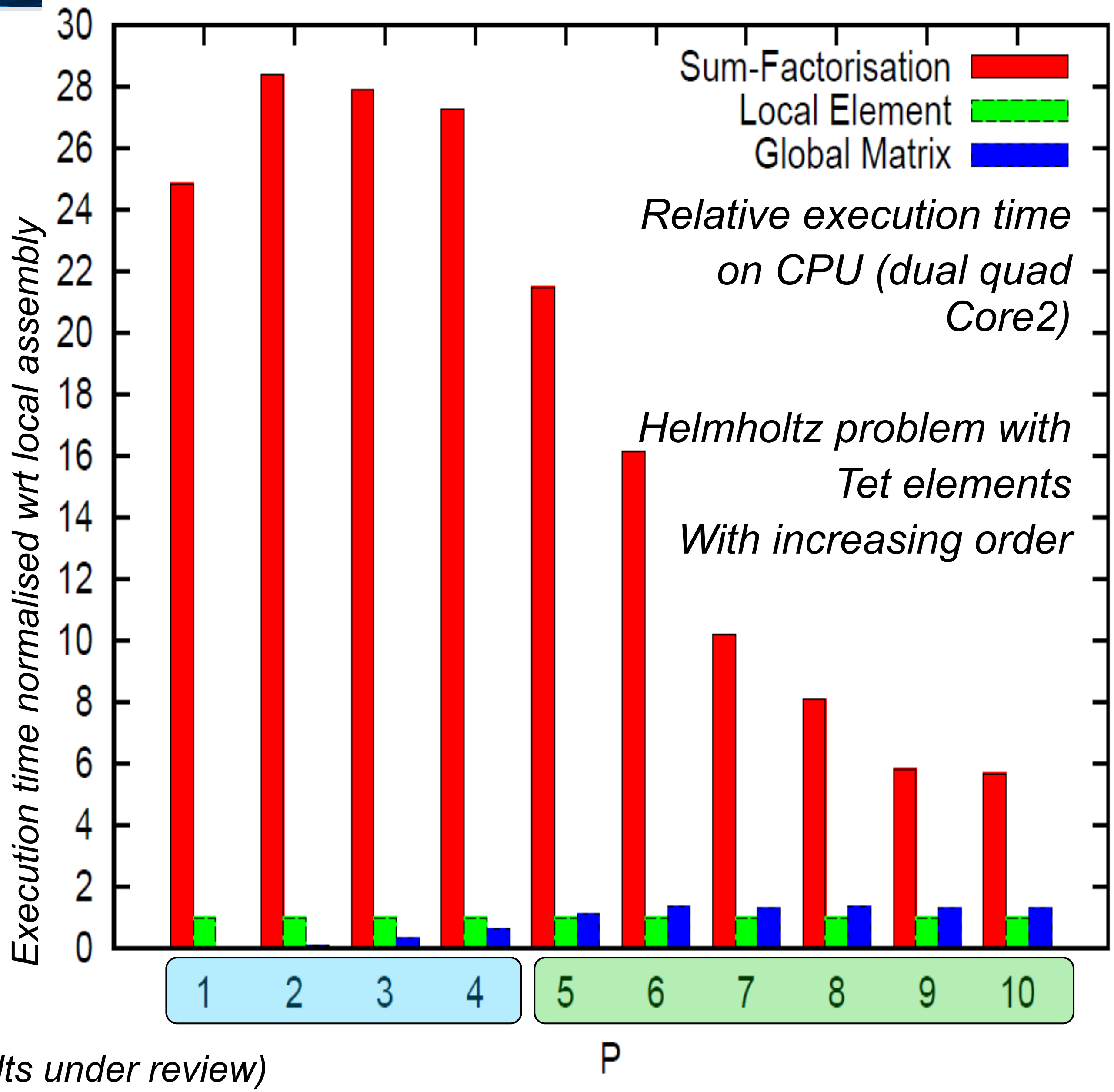


(Cantwell et al, provisional results under review)

Imperial College London

Mapping the design space – h/p

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

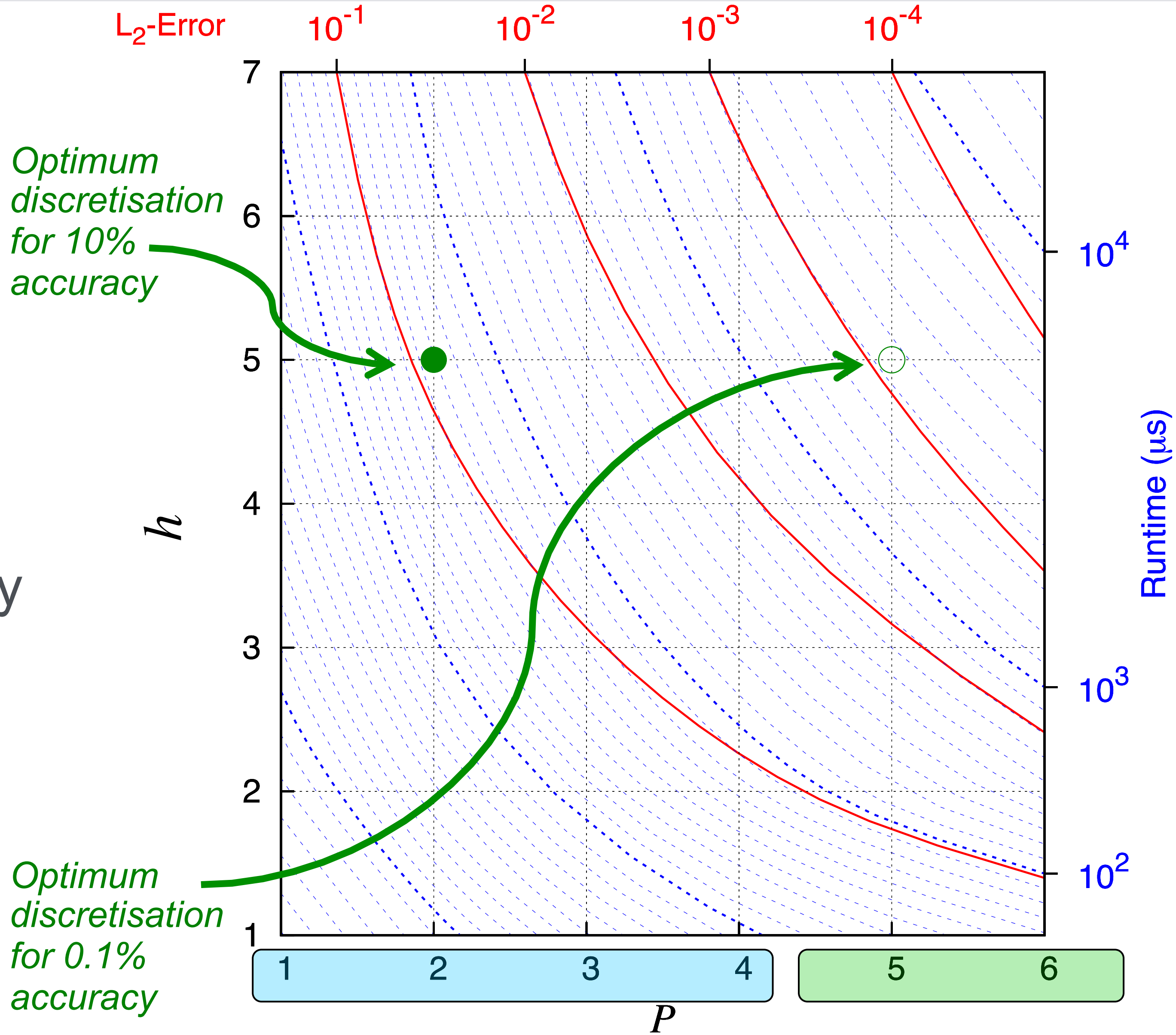


(Cantwell et al, provisional results under review)

Imperial College London

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

A roadmap: taking a vertical view

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

Transformation and optimisation framework

OpenCL

Other back-ends, eg AVX intrinsics, FPGAs

nVidia Fermi

Intel Larrabee

AMD GPUs

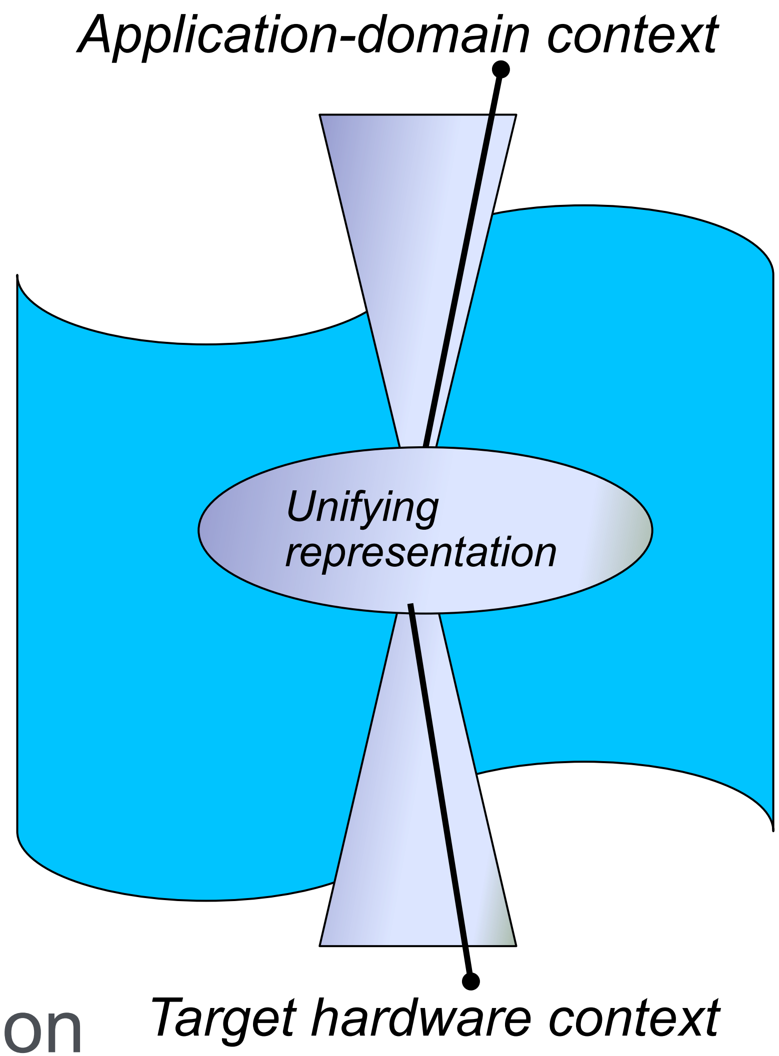
AVX

Future platforms

Imperial College London

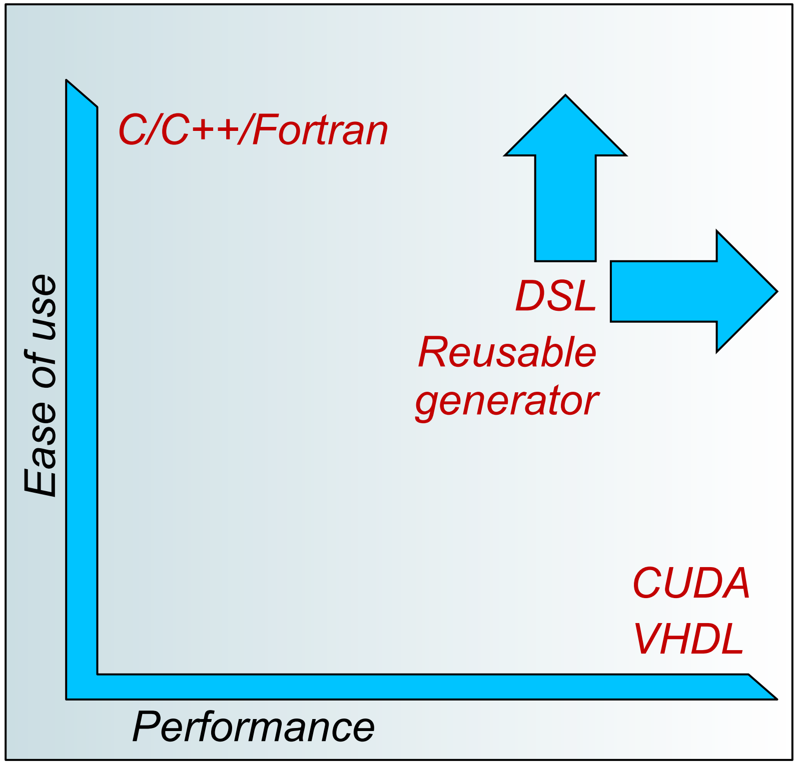
Conclusions and Further Work

- 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



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



Imperial College London

Acknowledgements

- Thanks to Lee Howes, Ben Gaster and Dongping Zhang at AMD

- 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