

# **(Towards) Generating optimised finite element solvers for GPUs**

**Graham Markall<sup>1</sup>, Andras Slemmer<sup>1</sup>, David Ham<sup>2</sup> and Paul Kelly<sup>1</sup>**

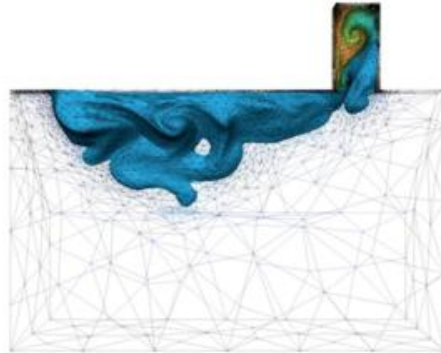
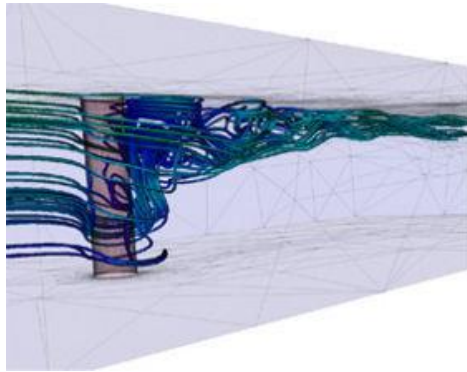
**1. Software Performance Optimisation Group  
Department of Computing  
Imperial College London**

**2. Earth Science and Engineering/  
Grantham Institute for Climate Change  
Imperial College London**

**GPUS and Accelerators in HPC Workshop  
at Daresbury Laboratory  
28 September 2010**

<http://www.doc.ic.ac.uk/~grm08/>  
[grm08@doc.ic.ac.uk](mailto:grm08@doc.ic.ac.uk)

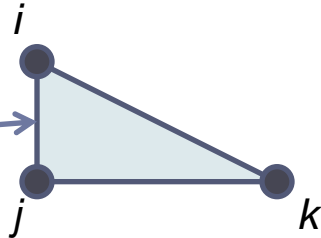
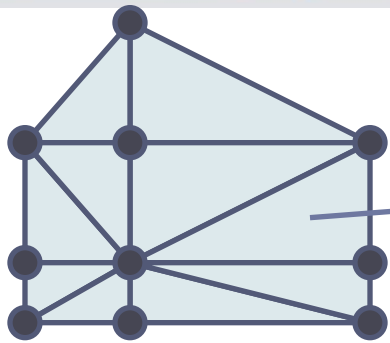
- **Fluidity** – Finite Element Adaptive Unstructured Mesh CFD Code
  - Used in Imperial College Ocean Model (ICOM)
  - Mainly Fortran sources, ~400kloc



- How do we exploit GPUs to improve the performance of Fluidity?
  - Write Fluidity again, in CUDA/OpenCL?
  - Build a CUDA/OpenCL-accelerated library covering key Fluidity components?
  - Use a **high-level Domain-Specific Language (DSL)** for describing the computation, and use **code generation**?
  - - Create code generation tools and get them into the hands of **real users**

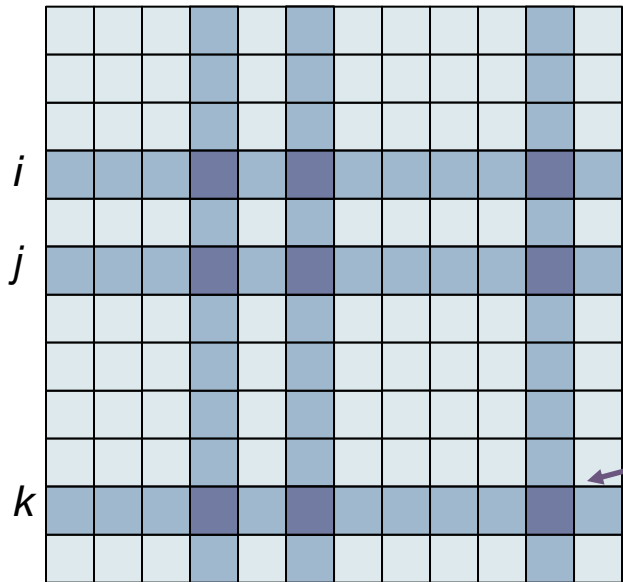
- Using a DSL opens up code generation choices that make a big difference in performance:
  - Program structure
  - Data structures & layout in memory
  - Algorithmic choices
- Ongoing work: code generator, incorporating optimisations determined by manual investigation
- This talk is about an exploration of the implementation spaces within the target architectures
- Next: background on FE and the DSL

# The FE Method: computation overview

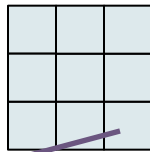


```
do element = 1, N
  assemble(element)
end do
```

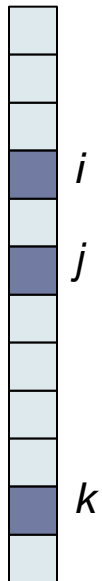
*i*   *j*   *k*



$$\int_{\Omega} v L(u^{\delta}) dX = \int_{\Omega} v q dX.$$



$$Ax = b$$



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

■ *PDE:*  $-\nabla^2 u = f, \quad \frac{\partial u}{\partial n} = 0$

■ *Weak form:*  $\int_{\Omega} \nabla v \cdot \nabla u \, dX = \int_{\Omega} v f \, dX$

- *From the FEniCS project [Logg & Wells, 2010]*
- *Toolchain for automated problem solving on CPU +MPI (Dolfin, FFC, UFL, UFC, ...)*

■ *UFL:*

```

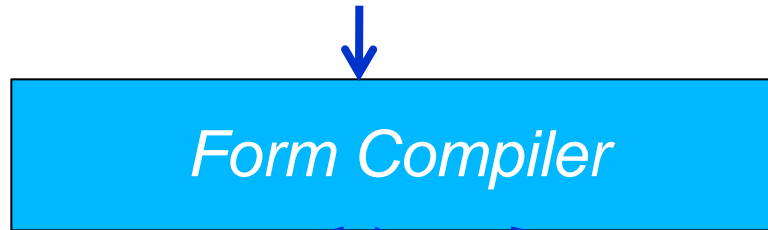
P=FiniteElement("Lagrange", "triangle", 1)

v=TestFunction(P); u=TrialFunction(P); f=Function(P)

A=dot(grad(v), grad(u)) * dx
RHS=v*f*dx
    
```

- *UFL allows a declarative specification of the problem to be given*
- *Does not commit to low-level implementation details*
- *Allows freedom in the choice of data structures and algorithms used in the final generated code*

## *Unified Form Language*



*Fortran/C++*

*OpenCL*

*CUDA*

*Multicore+SSE*

*ATI Cypress GPU*

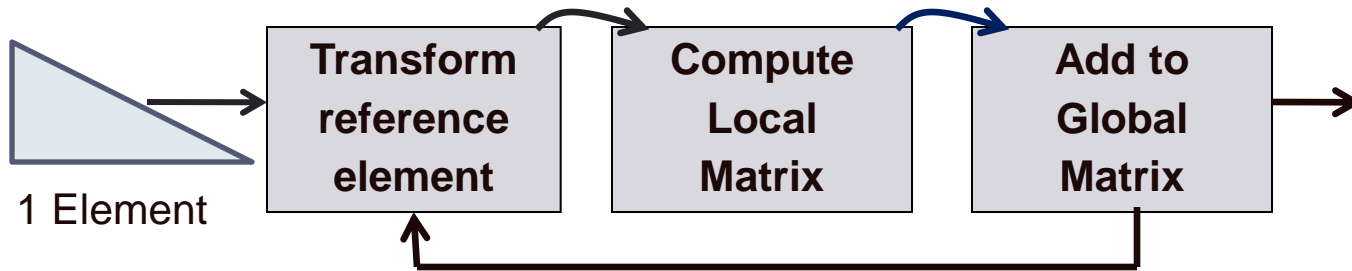
*Nvidia Fermi GPU*

- OpenCL allows us to run the same code on very diverse hardware
- Functional portability – but performance portability?

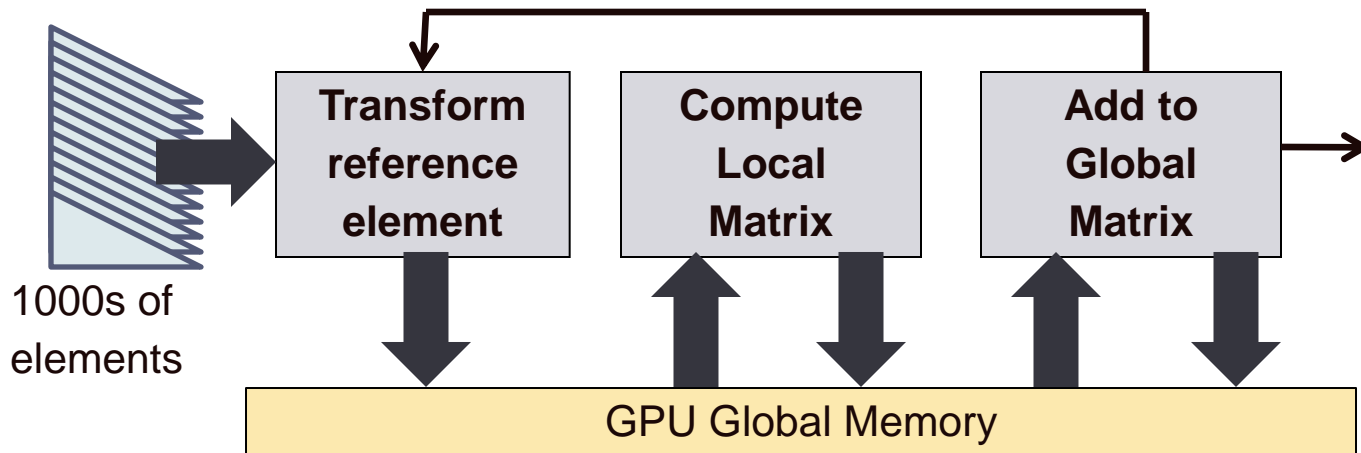
- GPU performance issues:
  - Needs massive parallelism
    - Many thousands of threads
    - Organised into hierarchy – grid, block, warp
    - Rendered as Multicore, SMT, and SIMD
  - Extremely small per-thread state
    - More registers per thread implies fewer threads, and less latency-hiding
    - Limited cache - very limited scope to exploit temporal locality
  - Extremely dependent on spatial data locality
    - Across the threads in a SIMD warp
  - Also dependent on spatial *branch* locality:
    - Across the threads in a SIMD warp

# Assembly Loop Structures

CPU (sequential, one instance per core):

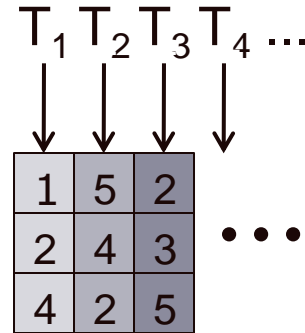
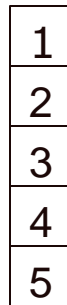
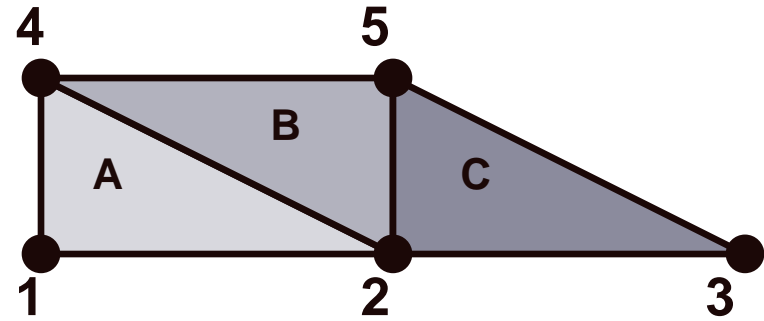
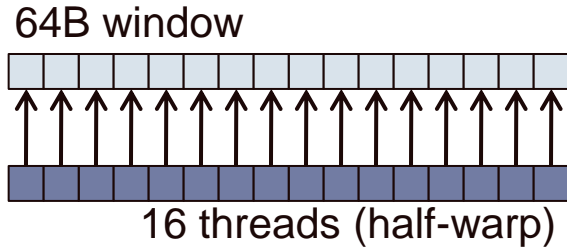


GPU (data parallel, one instance per device):



- For GPUs we have to split the assembly loop into kernels with controlled requirements for registers and cache/scratchpad capacity

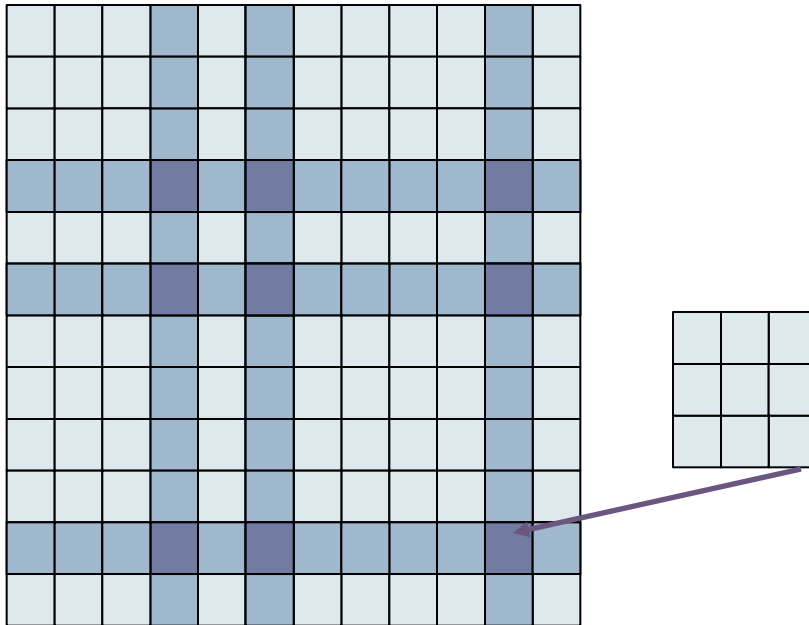
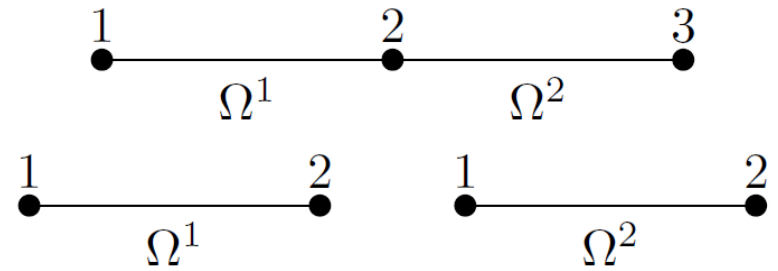
Coalescing (window size varies):



**Coalescing, destroys temporal locality**  
**Expansion 2D: 6, 3D: 24**

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

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



*The Addto Algorithm*

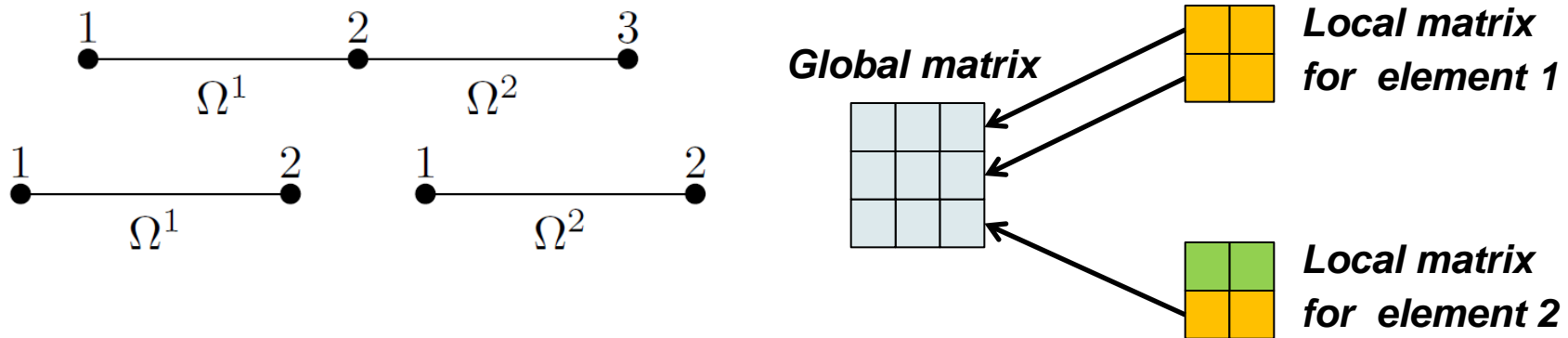
$$M^e = \begin{bmatrix} m_{11}^1 & m_{12}^1 & & & & \\ m_{21}^1 & m_{22}^1 & & & & \\ & & & & & \\ & & & m_{11}^2 & m_{12}^2 & \\ & & & m_{21}^2 & m_{22}^2 & \end{bmatrix}$$

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

- Combining the local assembly matrices into a sparse global system matrix

Parallelising the global assembly leads to performance/correctness issues:

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



- Set 0
- Set 1

- Is there a better way to do the global assembly?

- Why do we assemble  $M$ ?

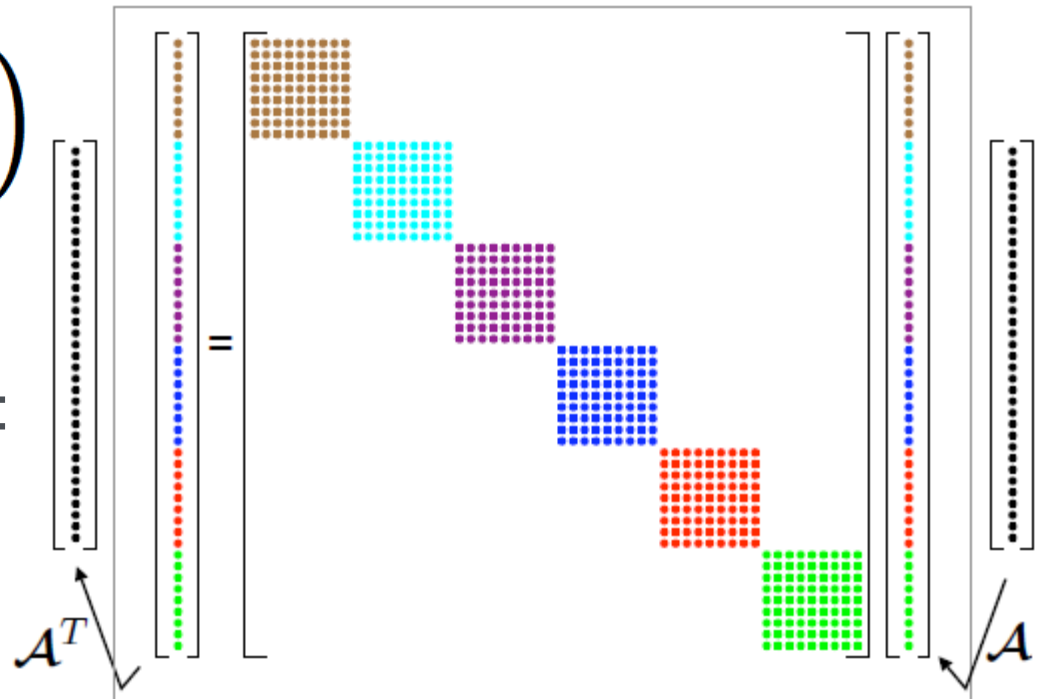
$$y = Mv \quad \text{where} \quad M = A^T M^e A$$

- Derive the *Local Matrix Approach*:

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

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

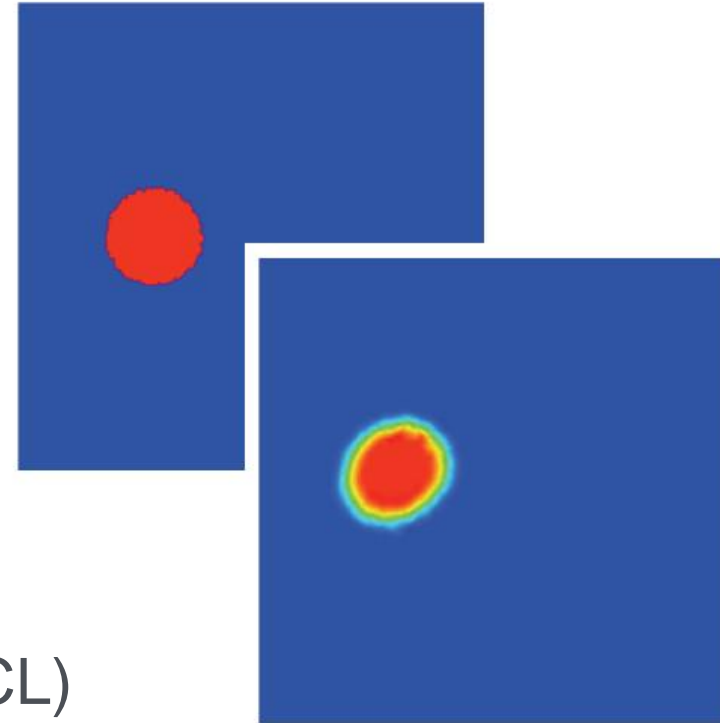
$$b = A^T b^e$$



## ■ Advection-Diffusion Equation:

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

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



## ■ 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*

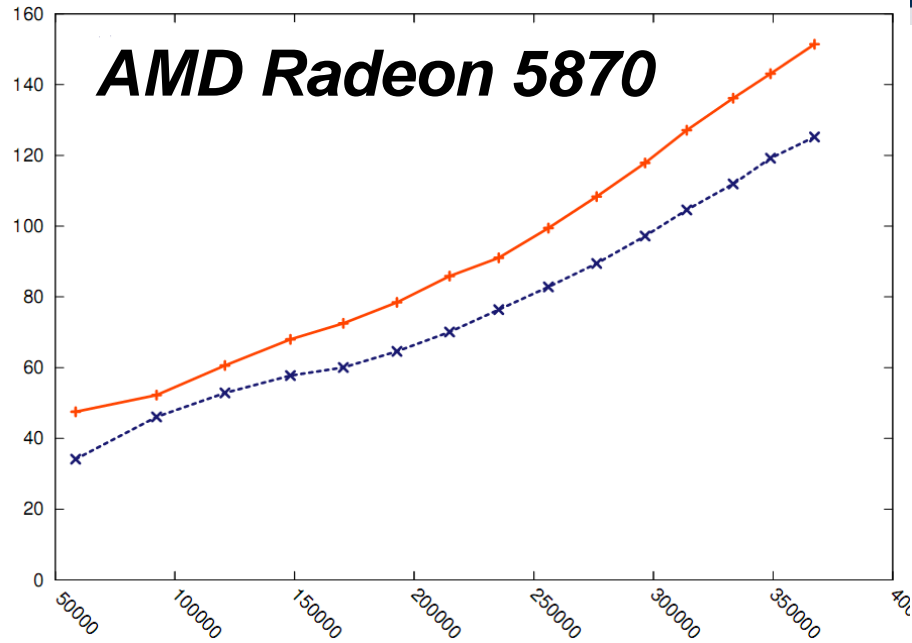
*ATI Stream SDK 2.2 for OpenCL*

### **Linear Solver:**

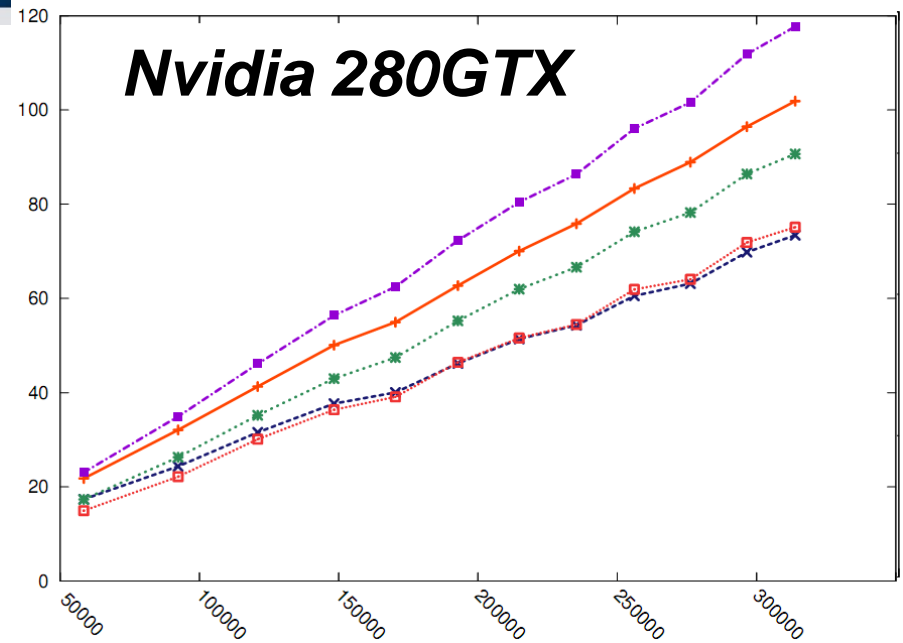
*CPU: PETSc [Balay et al., 2010]*

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

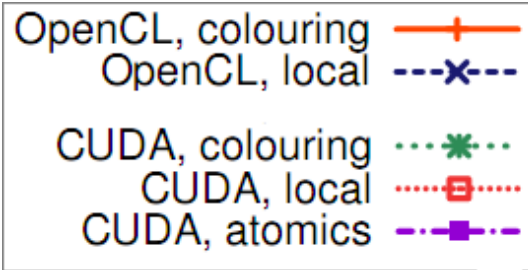
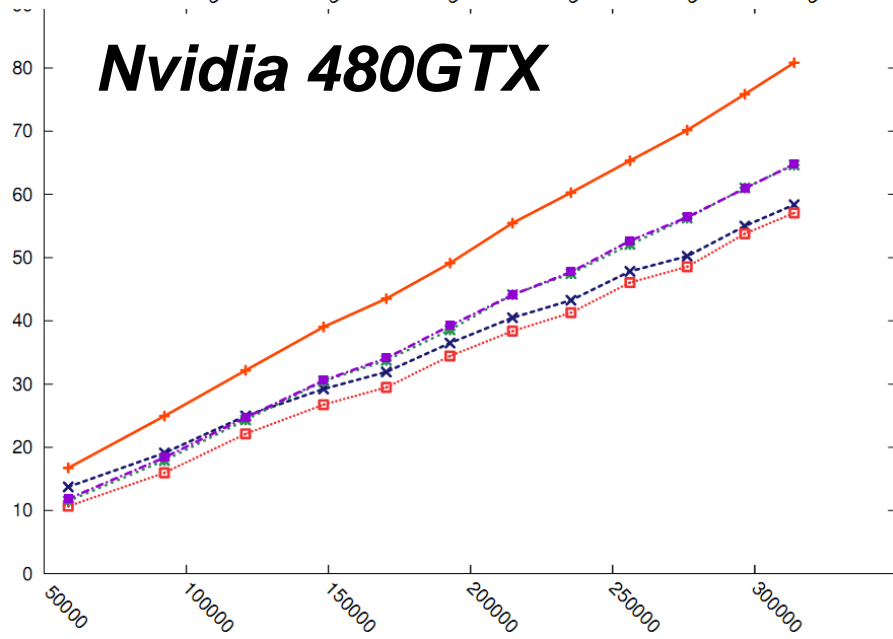
## AMD Radeon 5870



## Nvidia 280GTX



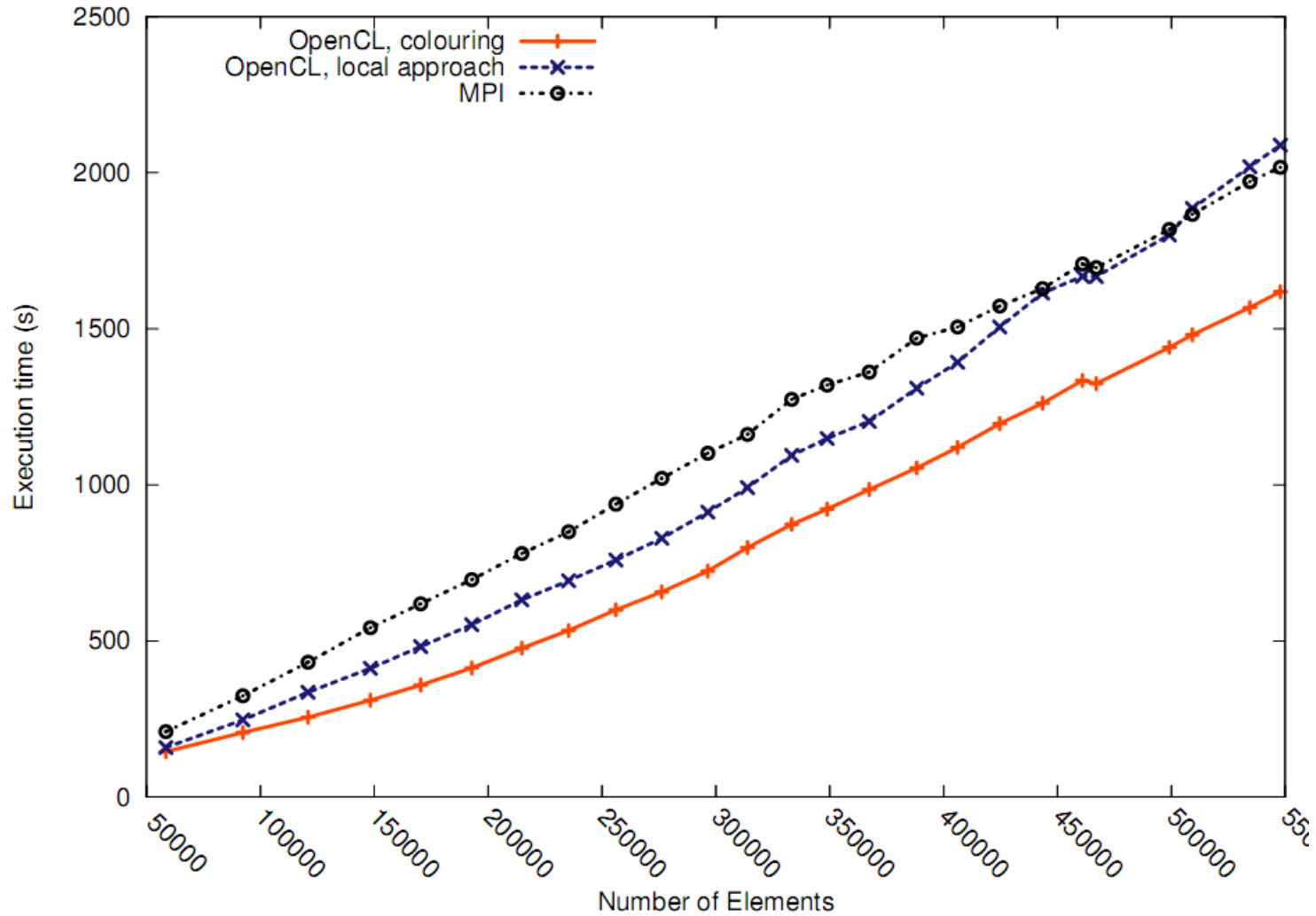
## Nvidia 480GTX



**X axis:** no. elements

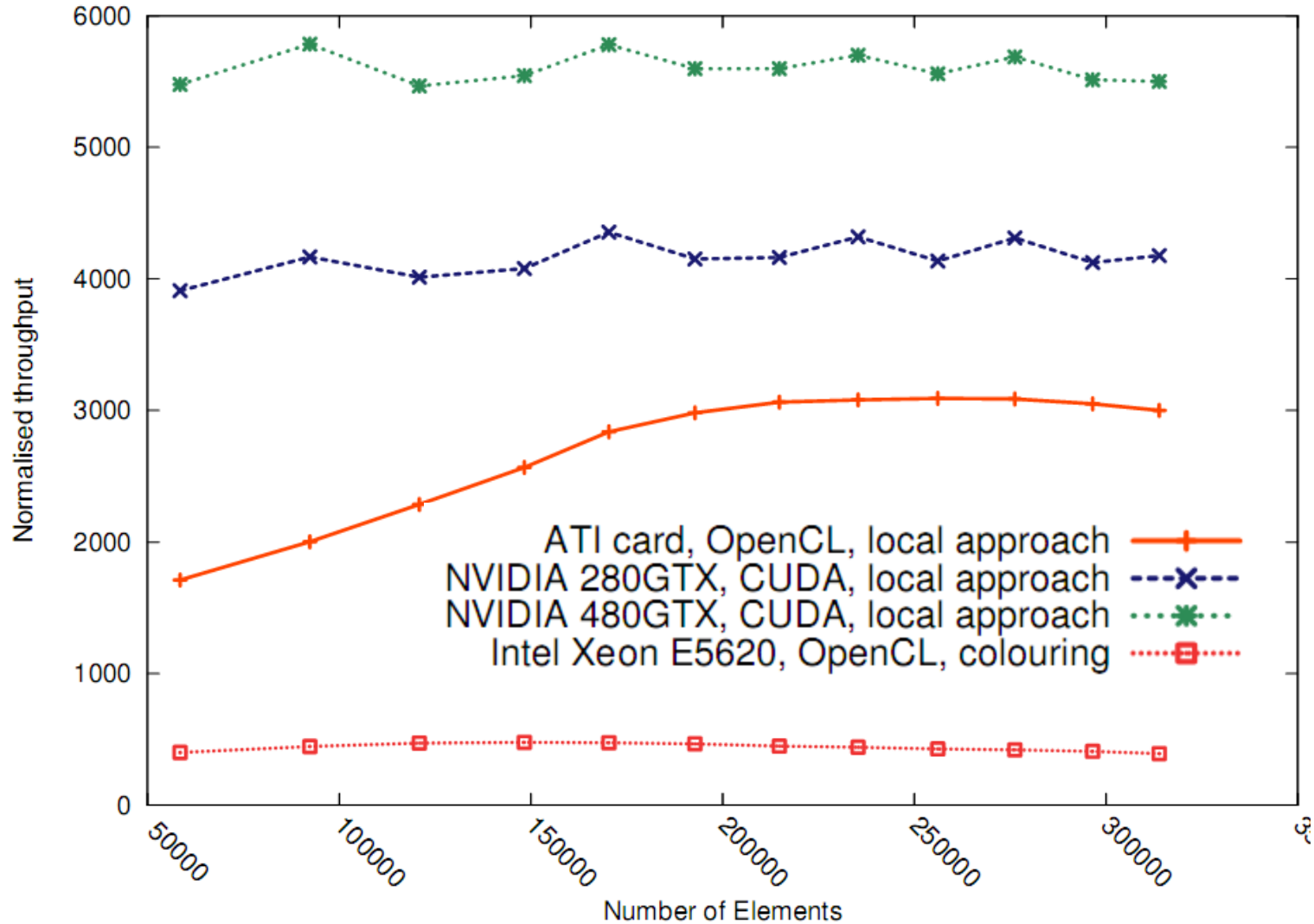
**Y axis:** simulation time (s)

**Trend:** LMA faster than Addto



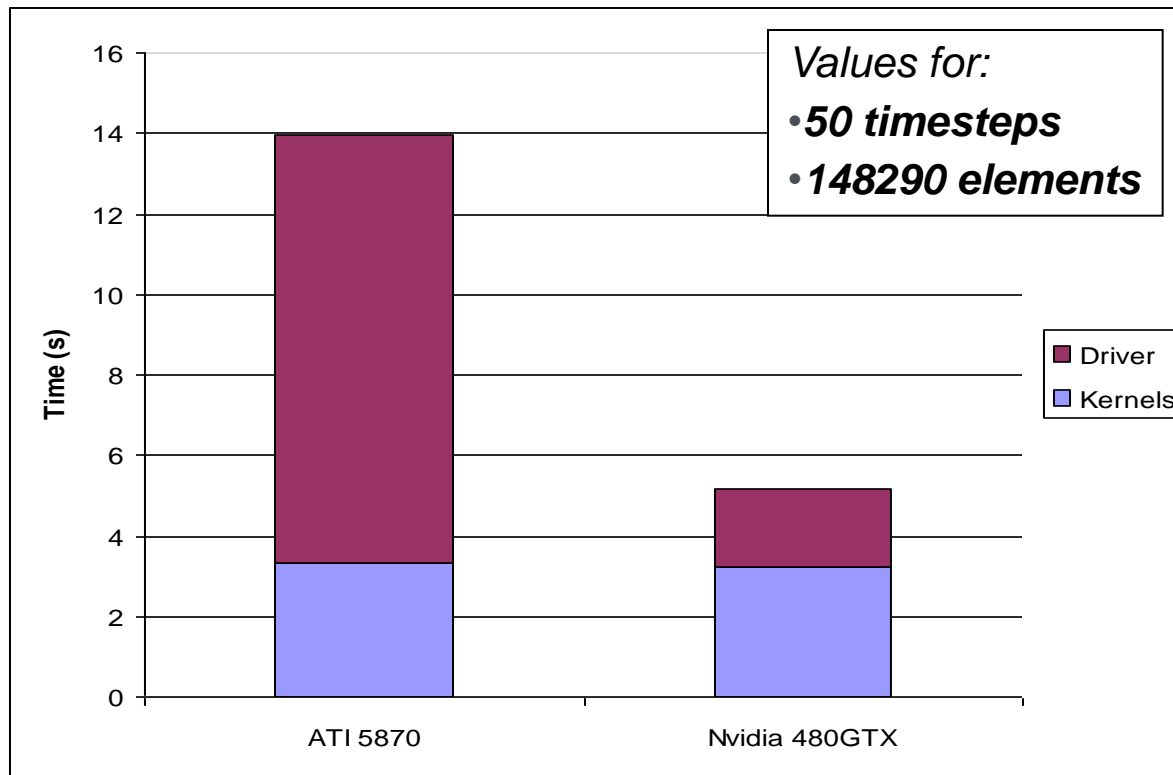
**Trend:** Addto is faster than the local matrix approach

# Relative performance



■ Order of magnitude speedups from current GPU hardware over current CPU hardware

- Performance of AMD 5870 lower than expected
- Events and Profiling information for measuring kernel execution time:



- Kernel execution times similar on both cards
- Overall times different due to large overhead in AMD drivers

- 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

- From these experiments:
  - Algorithm choice makes a big difference in performance
  - The best choice varies with the target hardware
- With automated code generation:
  - We can navigate the design space freely
  - And pick the best implementation strategy for each context

**[Balay et al., 2010]** Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Web page, 2009.

<http://www.mcs.anl.gov/petsc>.

**[Filipovic et al., 2009a]** Jiri Filipovic, Igor Peterlik, and Jan Fousek. GPU Acceleration of Equations Assembly in Finite Elements Method - Preliminary Results. Symposium on Application Accelerators in HPC, July 2009.

**[Geuzaine and Remacle, 2009]** Geuzaine, C. and Remacle, J.F., Gmsh: a three-dimensional finite element mesh generator with built-in pre-and post-processing facilities, International Journal for Numerical Methods in Engineering, 79(11):1309-1331, 2009.

**[Klockner et al., 2009]** A. Klockner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. Journal of Computational Physics, In Press, 2009.

**[Logg, 2007]** A. Logg. Automating the finite element method. Arch. Comput. Methods Eng., 14(2):93–138, 2007.

**[Markall and Kelly, 2009]** Graham Markall and Paul H. J. Kelly. Accelerating Unstructured Mesh Computational Fluid Dynamics Using the NVidia Tesla GPU Architecture. ISO Report, Imperial College London, 2009.

**[NVidia, 2010]** NVidia. NVIDIA CUDA Programming Guide.

[http://developer.download.nvidia.com/compute/cuda/3\\_0/toolkit/docs/NVIDIA\\_CUDA\\_ProgrammingGuide.pdf](http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf) , Retrieved 06 May 2010.

**[Vos et al., 2010]** Peter E. J Vos, Spencer J. Sherwin, and Robert 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.

<http://www2.imperial.ac.uk/ssherw/spectralhp/papers/JCP-VoShKi-09.pdf>, October 2009.



- *Domain-specific languages*
  - *Raise the level of abstraction*
  - *Capture a domain of variability*
  - *Encapsulate reuse of a body of code generation expertise/technology*
  - *Enable us to capture a design space*
    - *To match implementation choice to application context*
  - *Context:*
    - *Target hardware*
    - *Problem instance*
- *This talk aims to illustrate these ideas with our recent work towards automatic generation of multicore and manycore code for the assembly phase of the finite element method*

# Active libraries

- Domain-specific “active” library encapsulates specialist performance expertise

Visual effects

Finite element

Linear algebra

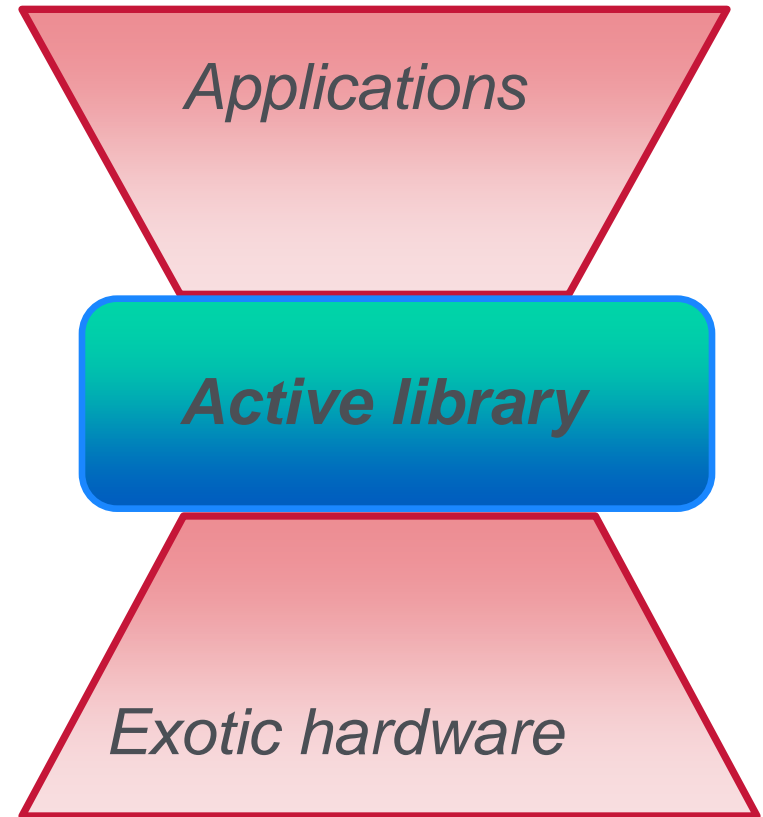
Game physics

Finite difference

- Each new platform requires new performance tuning effort

- So domain-specialists will be doing the performance tuning

- Our challenge is to support them



GPU

Multicore

FPGA

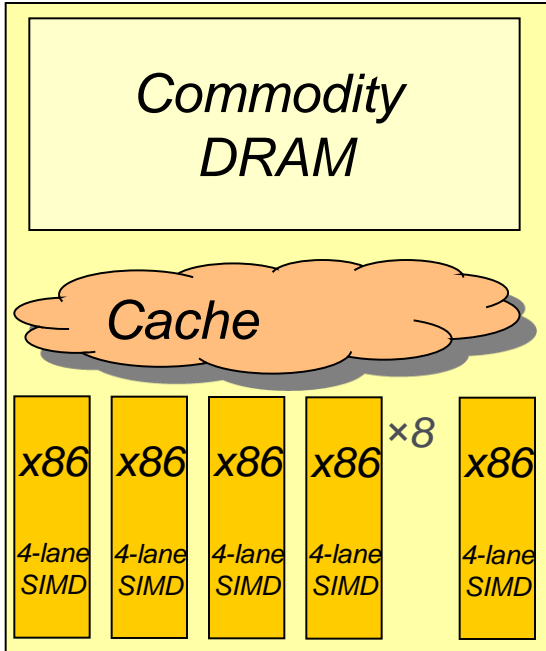
Quantum?

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

```
__global__ void form( ... )
{
    for(int ele=THREAD_ID; ele<n_ele; ele+=THREAD_COUNT)
    {
        for(int i=0; i<n_basis_fns; i++)
        {
            for(int j=0; j<n_basis_fns; i++)
            {
                localMatrix[idx(ele,i,j)] = 0.0;

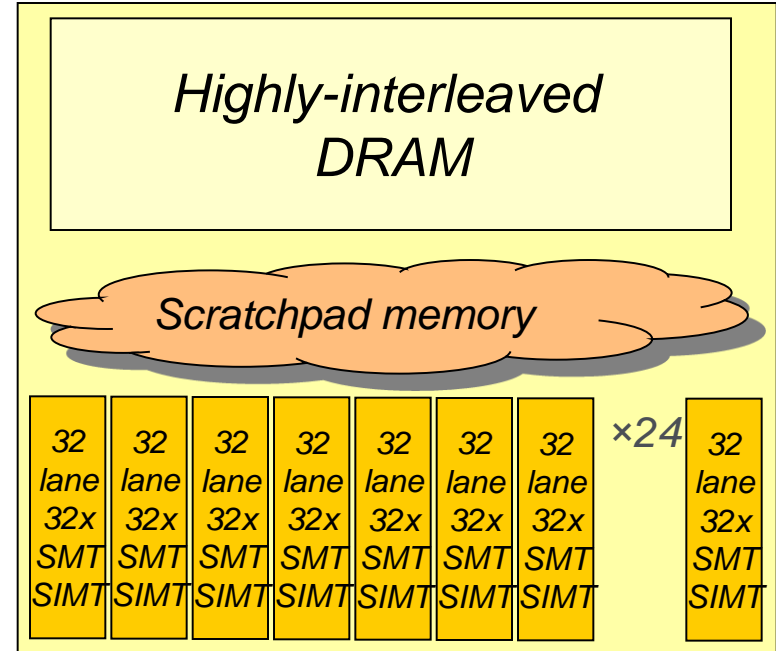
                for(int gauss_pt=0; gauss_pt<n_gauss_pts; gauss_pt++)
                {
                    for(int dim=0; dim<n_dim; dim++)
                    {
                        localMatrix[idx(i,j)] += d_test_fn[idx(ele,i,gauss_pt,dim)]
                                                * d_trial_fn[idx(ele,j,gauss_pt,dim)]
                                                * quad_weight_J[idx(ele, gauss_pt)];
                    }
                }
            }
        }
    }
}
```

# Diversity in target hardware



- Lots of cache per thread
- Lower DRAM bandwidth

*SIMD Multicore CPU*



- Very, very little cache per thread
- Very small scratchpad RAM shared by blocks of threads
- Higher DRAM bandwidth

*SIMT Manycore GPU*