OP2: an active library for unstructured grid applications on GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford eResearch Centre

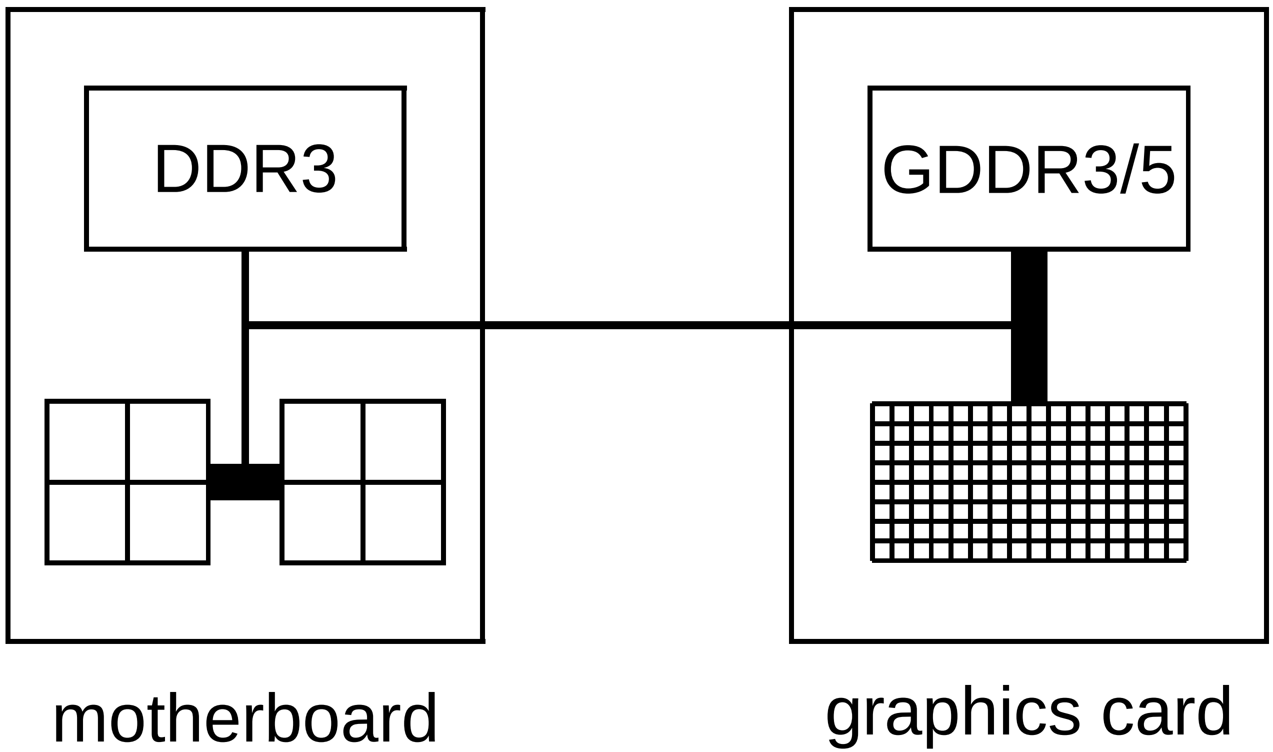
Seminar at Warwick University, May 7th, 2010

Outline

- programming on GPUs
- PDE applications – opportunity, challenges, context
- user perspective (i.e. application developer)
  - API
  - build process
- implementation issues
  - hierarchical parallelism on GPUs
  - data dependency
  - code generation
- current status
- lessons learned so far

GPU hardware

Typically, a PCIe graphics card with a many-core GPU sits inside a PC/server with one or two multicore CPUs:



GPU hardware

- CPUs have up to 6 cores (each with a SSE vector unit) and 10-30 GB/s bandwidth to main system memory
- current NVIDIA GPUs have up to 30×8 cores on a single chip and 100+ GB/s bandwidth to graphics memory
- offer 50–100$\times$ speedup relative to a single CPU core
- roughly 10$\times$ speedup relative to two quad-core Xeons
- also 10$\times$ improvement in price/performance and energy efficiency

How is this possible? Much simpler cores (SIMD units, no out-of-order execution or branch prediction) designed for vector computing, not general purpose

Emergence of GPUs

- AMD, IBM (Cell) and Intel (Larrabee) all producing or developing GPUs too
- NVIDIA has a good headstart on software side with CUDA environment
- new OpenCL software standard (based on CUDA and pushed by Apple) will probably run on all platforms
- driving applications are:
  - computer games “physics”
  - video (e.g. HD video decoding)
  - computational science
  - computational finance
  - oil and gas

Why GPUs will stay ahead?

Technical reasons:

- SIMD units means larger proportion of chip devoted to floating point computation (but CPUs will respond with longer vector units – AVX)
- tightly-coupled fast graphics memory means much higher bandwidth

Commercial reasons:

- CPUs driven by price-sensitive office/home computing; not clear these need vastly more speed
- CPU direction may be towards low cost, low power chips for mobile and embedded applications
- GPUs driven by high-end applications – prepared to pay a premium for high performance

Programming

Big breakthrough in GPU computing has been NVIDIA's development of CUDA programming environment

- C plus some extensions and an increasing number of C++ features
- host code runs on CPU, CUDA code runs on GPU
- explicit movement of data across the PCIe connection
- difficulty of programming is comparable to OpenMP plus MPI; fairly short learning curve for those with experience with both
- future hardware will make the programming easier by providing L1/L2 caches

Opportunity and Challenge

- PDE applications are of major importance in both academia and industry
- new HPC hardware (GPUs, AVX, etc.) offers 10$\times$ improvement in performance of affordable HPC but greatly increased programming complexity
- want a suitable level of abstraction to separate the user's specification of the application from the details of the parallel implementation
- aim to achieve code longevity and near-optimal performance through re-targetting the back-end to different hardware

Context

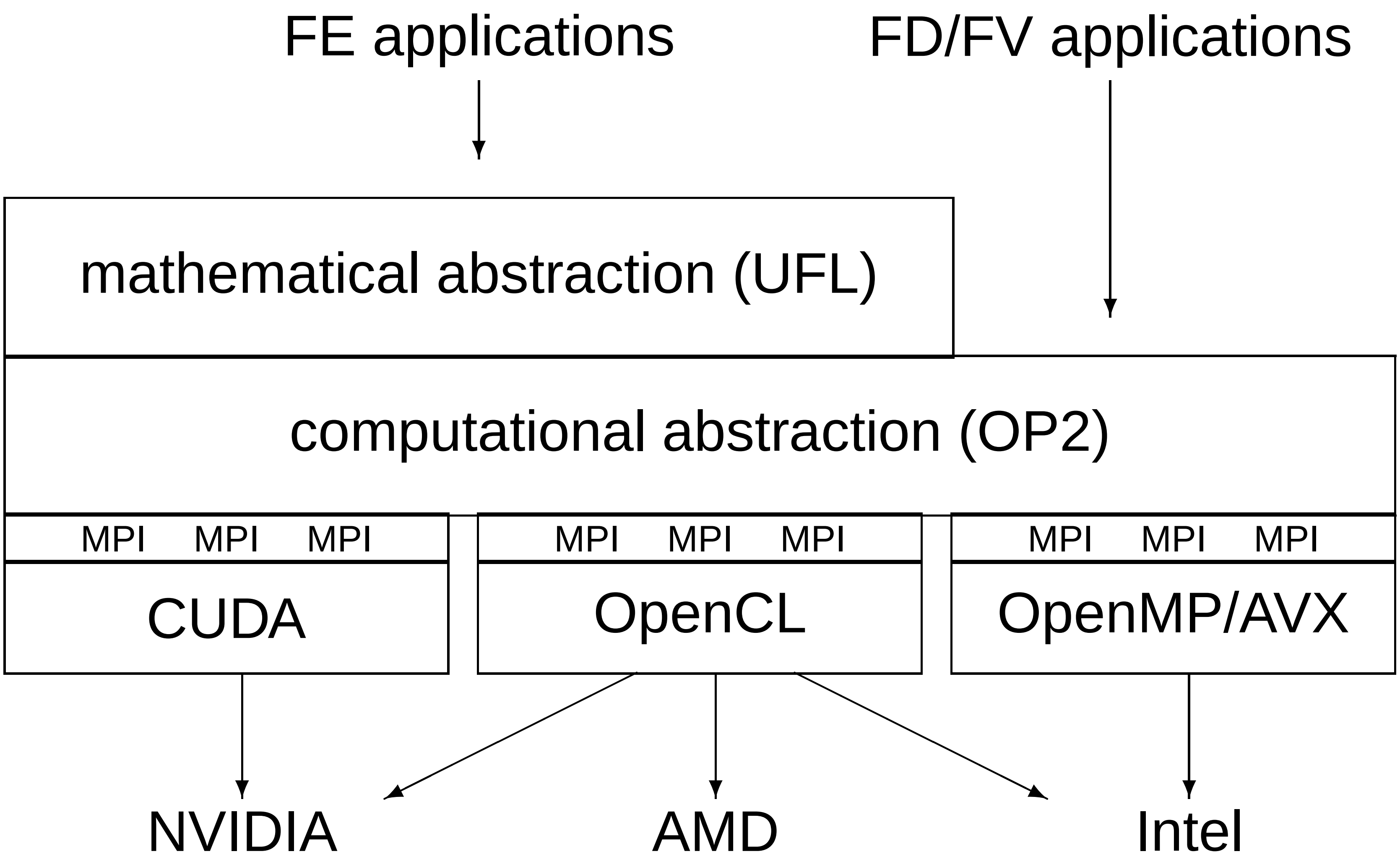
Unstructured grid methods are one of Phil Colella's seven dwarfs (*Parallel Computing: A View from Berkeley*)

- dense linear algebra
- sparse linear algebra
- spectral methods
- N-body methods
- structured grids
- unstructured grids
- Monte Carlo

Extensive GPU work for the other dwarfs, except perhaps for direct sparse linear algebra.

Context

Part of a larger project led by Paul Kelly at Imperial College



History

OPlus (Oxford Parallel Library for Unstructured Solvers)

- developed for Rolls-Royce 10 years ago
- MPI-based library for HYDRA CFD code on clusters with up to 200 nodes

OP2:

- open source project
- keeps OPlus abstraction, but slightly modifies API
- an “active library” approach with code transformation to generate CUDA, OpenCL and OpenMP/AVX code for GPUs and CPUs

OP2 Abstraction

- sets (e.g. nodes, edges, faces)
- datasets (e.g. flow variables)
- pointers (e.g. from edges to nodes)
- parallel loops
  - operate over all members of one set
  - datasets have at most one level of indirection
  - user specifies how data is used
    (e.g. read-only, write-only, increment)

OP2 Restrictions

- set elements can be processed in any order, doesn’t affect result to machine precision
  - explicit time-marching, or multigrid with an explicit smoother is OK
  - Gauss-Seidel or ILU preconditioning in not
- static sets and pointers (no dynamic grid adaptation)

OP2 API

```
op_init(int argc, char **argv)

op_decl_set(int size, op_set *set, char *name)

op_decl_ptr(op_set from, op_set to, int dim,
            int *iptr, op_ptr *ptr, char *name)

op_decl_const(int dim, char *type,
            T *dat, char *name)

op_decl_dat(op_set set, int dim, char *type,
            T *dat, op_dat *data, char *name)

op_exit()
```

OP2 API

Parallel loop for user kernel with 3 arguments:

```
op_par_loop_3(void (*kernel)(T0*, T1*, T2*),
    char *name, op_set set,
    op_dat arg0, int idx0, op_ptr ptr0,
    int dim0, char *typ0, op_access acc0,
    op_dat arg1, int idx1, op_ptr ptr1,
    int dim1, char *typ1, op_access acc1,
    op_dat arg2, int idx2, op_ptr ptr2,
    int dim2, char *typ2, op_access acc2)
```

Example for sparse matrix-vector product:

```
op_par_loop_3(res,"res", edges,
      p_A, -1,OP_ID, 1,"float",OP_READ,
      p_u,  0,pedge2,1,"float",OP_READ,
      p_du, 0,pedge1,1,"float",OP_INC);
```

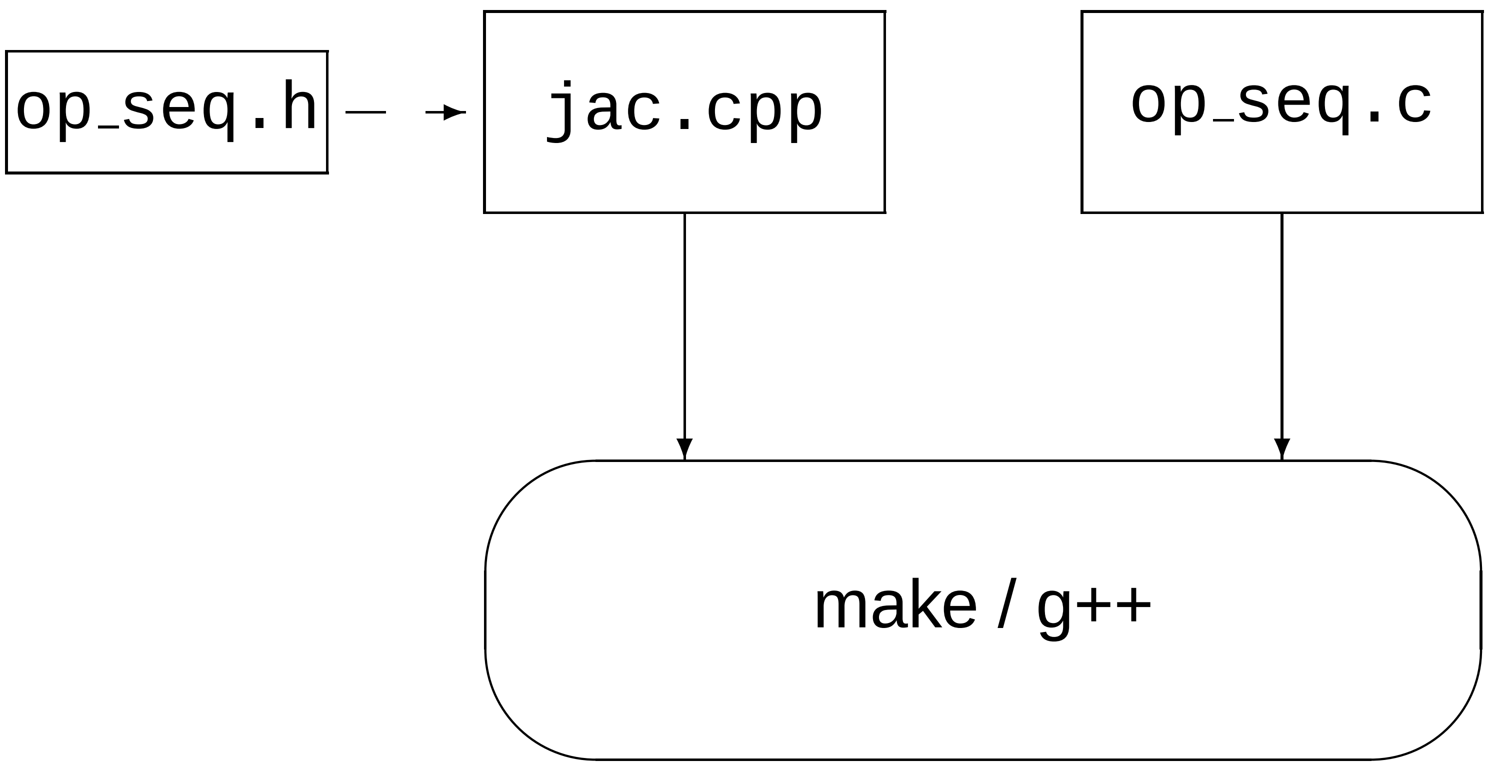
User build processes

Using the same source code, the user can build different executables for different target platforms:

- sequential single-thread CPU execution
  - purely for program development and debugging
  - very poor performance
- CUDA / OpenCL for single GPU
- OpenMP/AVX for multicore CPU systems
- MPI plus any of the above for clusters

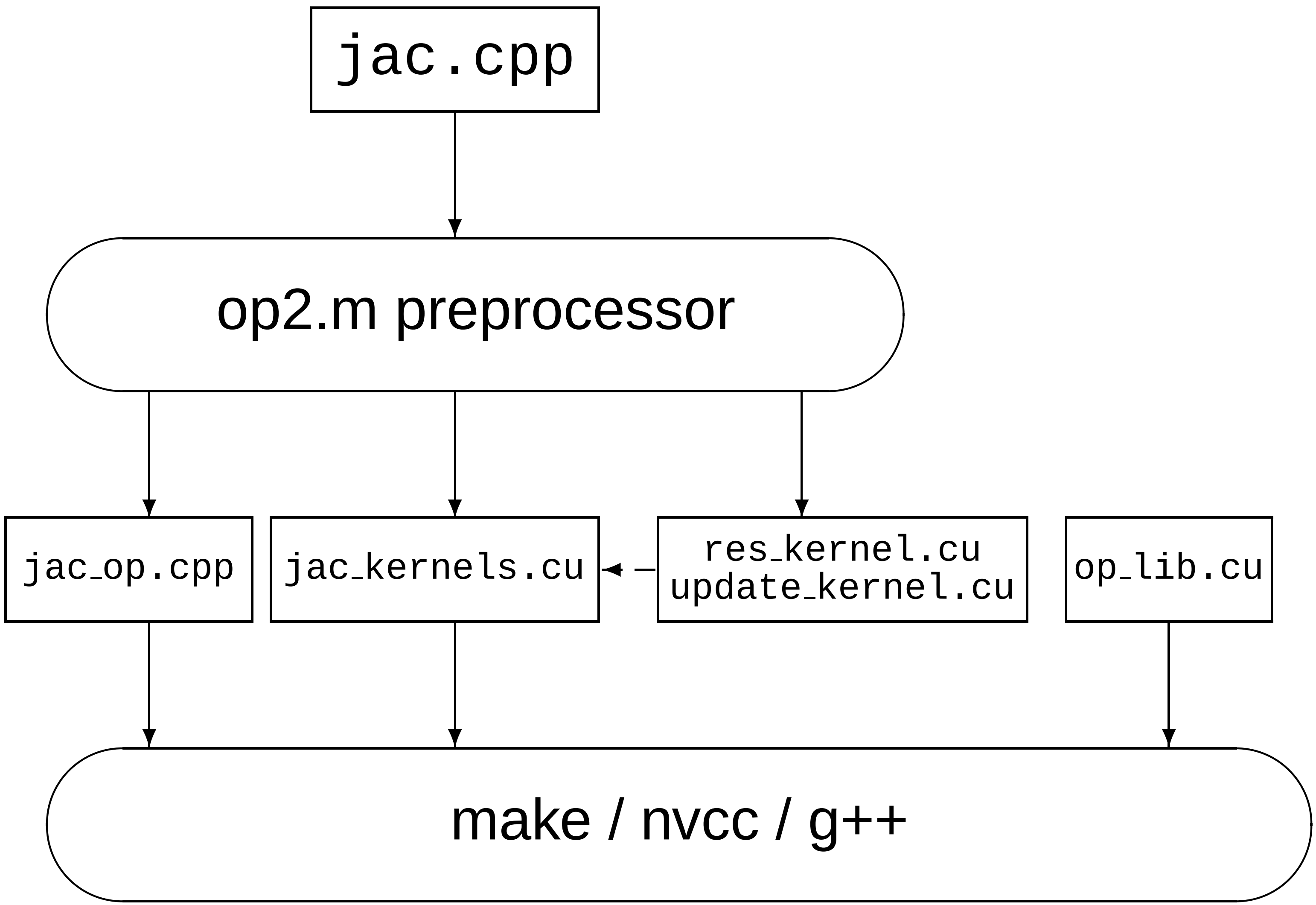
Sequential build process

Traditional build process, linking to a conventional library in which many of the routines do little but error-checking:



CUDA build process

Preprocessor parses user code and generates new code:



GPU Parallelisation

Could have up to 10^6 threads in 3 levels of parallelism:

- MPI distributed-memory parallelism (1-100)
  - one MPI process for each GPU
  - all sets partitioned across MPI processes, so each MPI process only holds its data (and halo)
- block parallelism (50-1000)
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different functional units in the GPU
- thread parallelism (32-128)
  - each mini-partition is worked on by a block of threads in parallel

GPU Parallelisation

The 16 functional units in an NVIDIA Fermi GPU each have

- 32 cores
- 48kB of shared memory
- 16kB of L1 cache

Mini-partitions are sized so that all of the indirect data can be held in shared memory and re-used as needed

- reduces data transfer from/to main graphics memory
- very similar to maximising cache hits on a CPU to minimise data transfer from/to main system memory
- implementation requires re-numbering from global indices to local indices – tedious but not difficult

GPU Parallelisation

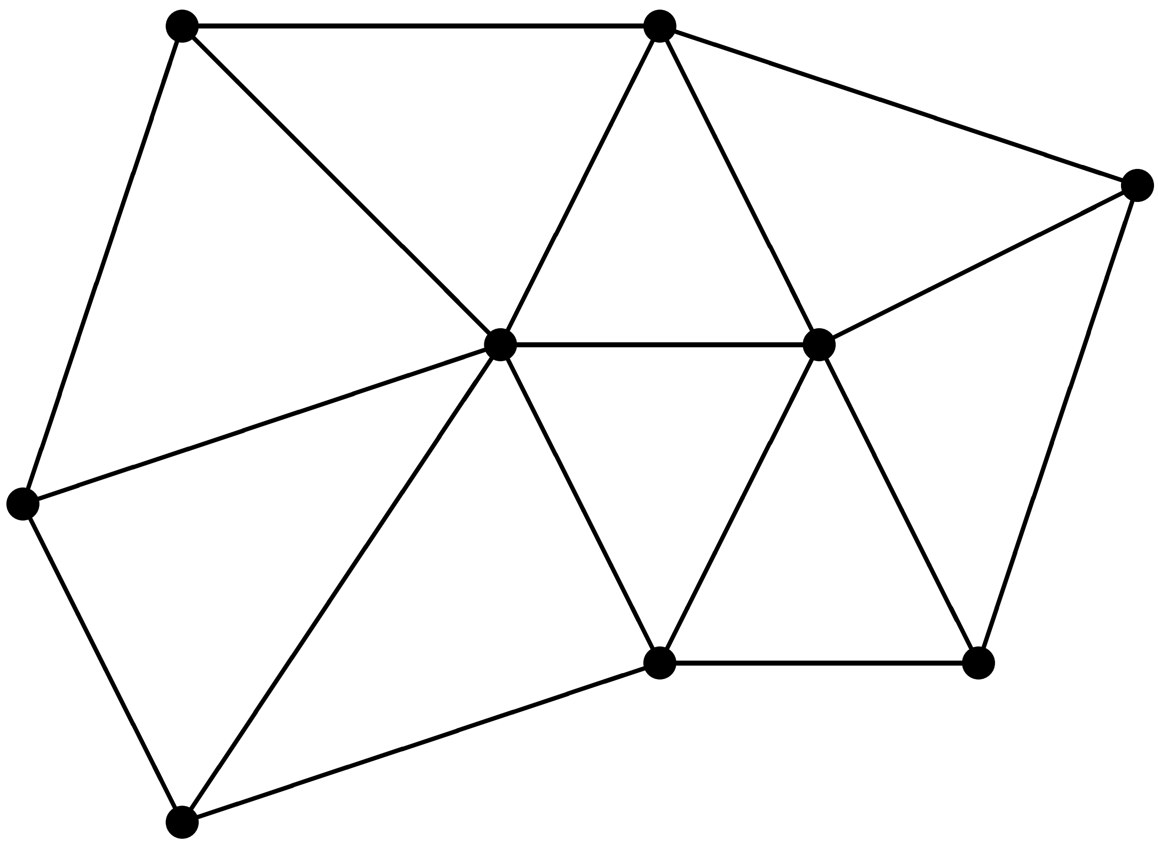
One important difference from MPI parallelisation

- when using one GPU, all data is held in graphics memory in between each parallel loop
- each loop can use a different set of mini-partitions
- current implementation constructs an “execution plan” the first time the loop is encountered
- auto-tuning will be used in the future to optimise the plan, either statically based on profiling data, or dynamically based on run-time timing

Data dependencies

Key technical issue is data dependency when incrementing indirectly-referenced arrays.

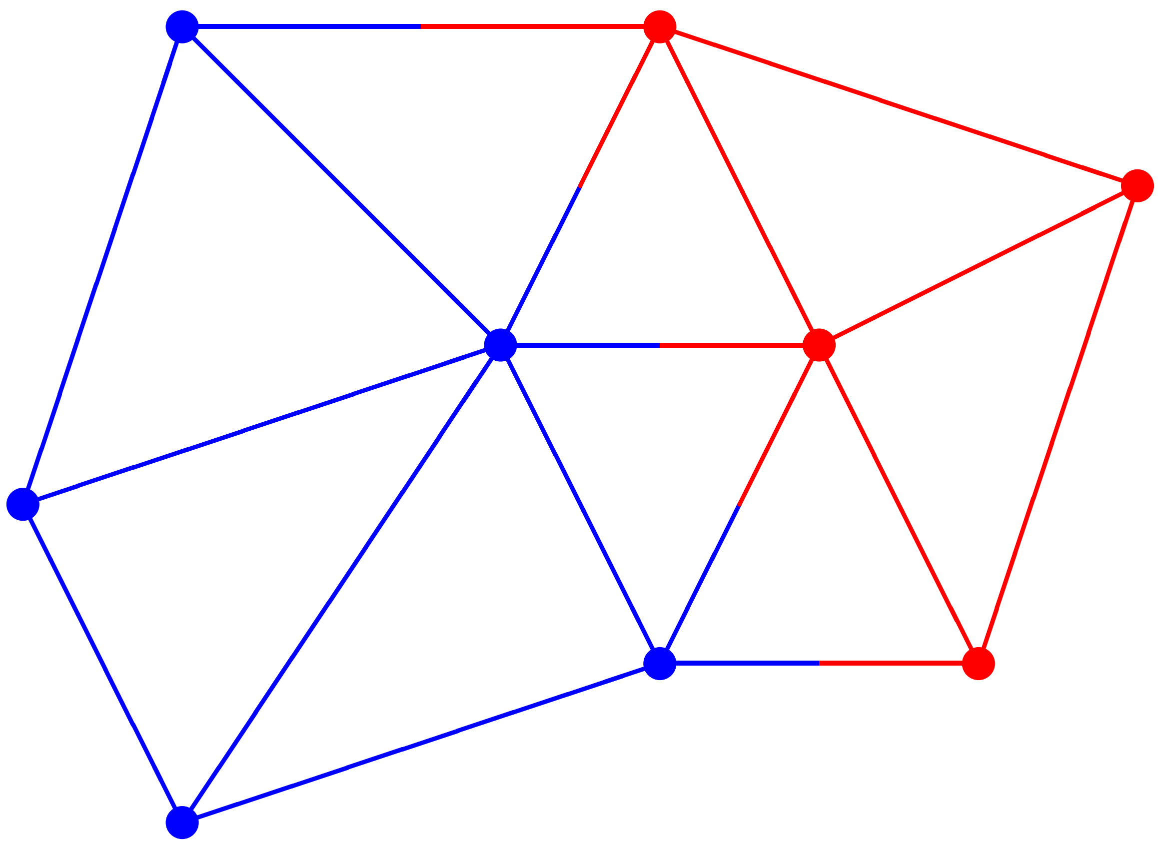
e.g. potential problem when two edges update same node



Data dependencies

Method 1: "owner" of nodal data does edge computation

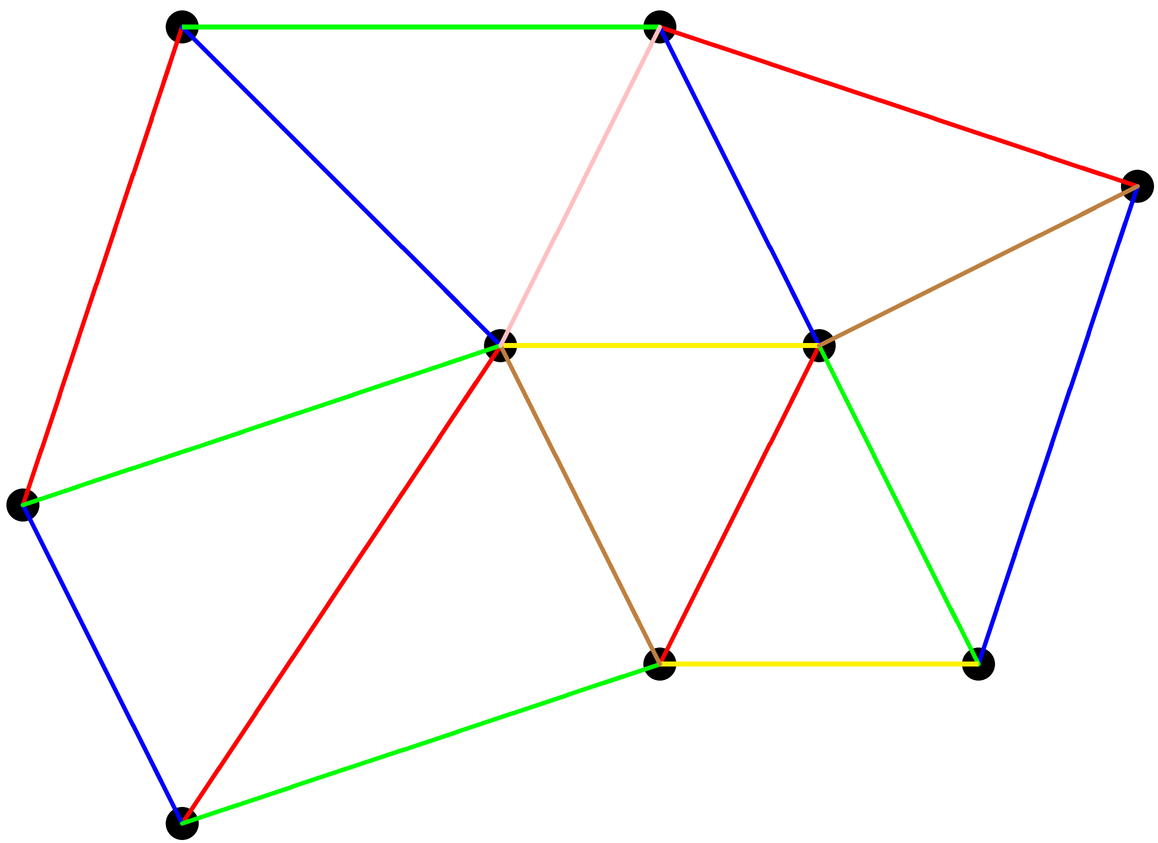
- drawback is redundant computation when the two nodes have different "owners"



Data dependencies

Method 2: “color” edges so no two edges of the same color update the same node

- parallel execution for each color, then synchronize
- possible loss of data reuse and some parallelism



Data dependencies

Method 3: use “atomic” add which combines read/add/write into a single operation

- avoids the problem but needs hardware support
- drawback is slow hardware implementation

	without atomics		with atomics	
	thread 0	thread 1	thread 0	thread 1
	read		atomic add	
time		read		
	add			atomic add
		add		
	write			
		write		

Data dependencies

Which is best for each level?

- MPI level: method 1
  - each MPI process does calculation needed to update its data
  - partitions are large, so relatively little redundant computation
- GPU level: method 2
  - plenty of blocks of each color so still good parallelism
  - data reuse within each block, not between blocks
- block level: method 2 or 3
  - indirect data in local shared memory, so get reuse
  - which costs more, local synchronization or atomic updates?

Current status

- working CUDA prototype for single GPU, with preprocessor written in MATLAB
- simple demo application working fine, but need to debug new 2D airfoil application
- key bit missing: global set renumbering to improve mini-partitions (related to partitioning for MPI level)
- plan to look PGI FORTRAN CUDA this summer, and maybe OpenCL in the future
- new PC with NVIDIA's new Fermi GPU arriving today? — expanded shared memory, L1/L2 caches and greatly improved double precision performance are all vital
- then performance testing will start!

Future plans

- EPSRC project with Paul Kelly will fund 2 post-docs:
  - Oxford: 2 year position to work primarily on adding MPI to OP2 for GPU clusters

    Approach will probably follow OPlus treatment, but I would like to have performance monitoring, optimisation and prediction for new systems
  - Imperial: 3 year position to work primarily on higher-level finite element abstraction, but also responsible for OpenMP/AVX backend for OP2
- Another post-doc at Imperial will be funded by Rolls-Royce/TSB and will probably work on CUDA FORTRAN side of OP2, and porting of HYDRA

Lessons learned so far

1) Code generation works, and it’s not too difficult!

- in the past I’ve been scared of code generation since I have no computer science background
- key is the routine arguments have all of the information required, so no need to parse the entire user code
- now helping a student develop a code generator for stochastic simulations in computational biology
  - a generic solver is inefficient – a “hand-coded” specialised implementation for one specific model is much faster
  - code generator takes in model specification and tries to produce “hand-coded” custom implementation
- I think this is an important trend for the future

Lessons learned so far

2) Working at the “bleeding edge” can have its difficulties:

- an early programming error led to my windows “dissolving” due to X-server corruption – no data protection within graphics memory
- porting from CUDA 2.2 to CUDA 3.0 hit problems with new compiler bugs which needed worksarounds – I’m told it’s fixed in CUDA 3.1.
- have had some cases where the GPUs “lock-up” and the system needs to be rebooted
- in general though, I’m impressed by how far the hardware and software has come in a relatively short time

Lessons learned so far

3) The thing which is now causing me most difficulty / concern is the limited number of registers per thread

- limited to about 50 32-bit registers per thread
- above this the data is spilled to L1 cache, but only 16kB of this so when using 256 threads only an extra 16 32-bit variables
- above this the data is spilled to L2 cache, which is 384kB but shared between all of the units in the GPU, so only an extra 48 32-bit variables
- the compiler can probably be improved, but also there are tricks an expert programmer can use
- points to the benefits of an expert framework which does this for novice programmers

Lessons learned so far

4) Auto-tuning is going to be important

- there are various places in the CUDA code where I have a choice of parameter values (e.g. number of threads, number of blocks, size of mini-partitions)
- there are also places where I have a choice of implementation strategy (e.g. thread coloring or atomic updates?)
- what I would like is a generic auto-tuning framework which will optimise these choices for me, given a reasonably small set of possible values
- as a first step, a undergraduate CS student is going to work with me on a 3rd year project on this

Lessons learned so far

5) This is fun and I think it will have impact!

- speedup offered by GPUs is really impressive
- intellectual challenge is in harnessing that potential, and making it available to others in a much simpler form

Conclusions

- have defined a high-level framework for parallel execution of algorithms on unstructured grids
- looks encouraging for providing ease-of-use, high performance, and longevity through new back-ends

Acknowledgements:

- Tobias Brandvik, Graham Pullan (Cambridge), Paul Kelly, Graham Markall (Imperial College), Nick Hills (Surrey)
- Jamil Appa, Pierre Moinier (BAE Systems), Leigh Lapworth (Rolls-Royce)
- Tom Bradley, Jon Cohen and others (NVIDIA)
- EPSRC, NVIDIA and Rolls-Royce for financial support