

OP2: an open-source library for unstructured grid applications

Mike Giles, Gihan Mudalige

`mike.giles@maths.ox.ac.uk`

Oxford University Mathematical Institute

Oxford e-Research Centre

Stanford University, November 17, 2010

NVIDIA, November 22, 2010

Outline

- 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

New heterogeneous hardware

For 10 years, 1995-2005, HPC was relatively simple:

- large clusters, with each node having 2 scalar CPUs
- MPI programming with FORTRAN / C / C++

Now things have become much more complicated:

- multi-core CPUs – up to 12 cores / 24 threads per CPU
- each core also has a vector unit – doubling in size to AVX very soon (and doubling again in near future)
- GPUs have up to 512 cores
- Best programming approach unclear:
 - MPI + OpenMP, or MPI + Ct for CPUs
 - CUDA for GPUs (and CPUs?)

Software Challenges

- HPC application developers want the benefits of the latest hardware but are very worried about the software development costs, and the level of expertise required
- status quo is not an option – running 24 MPI processes on a single CPU would give very poor performance, plus we need to exploit the vector units
- For GPUs, I'm happy with CUDA, but like MPI it's too low-level for many people
- For CPUs, MPI + OpenMP may be a good starting point, and PGI/CRAY are proposing OpenMP extensions which would support GPUs and vector units
- However, hardware is likely to change rapidly in next few years, and developers can not afford to keep changing their software implementation

Software Abstraction

To address these challenges, need to move to a suitable level of **abstraction**:

- separate the user's **specification** of the application from the details of the parallel **implementation**
- aim to achieve application level **longevity** with the top-level specification not changing for perhaps 10 years
- aim to achieve near-optimal **performance** through re-targetting the back-end implementation to different hardware and low-level software platforms

Context

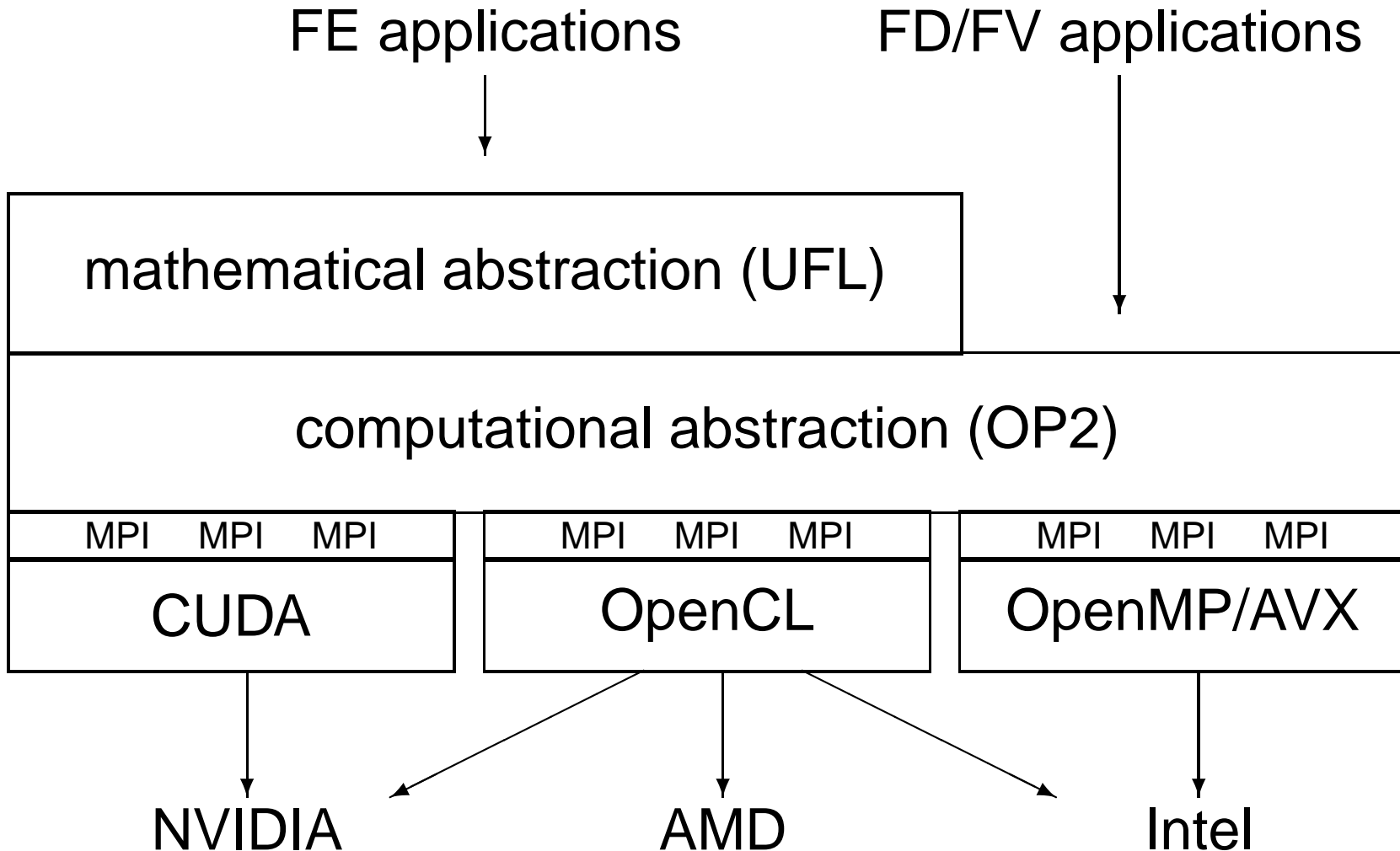
Unstructured grid methods are one of Phil Colella's seven dwarfs (*Parallel Computing: A View from Berkeley*)

- 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 is 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);
```

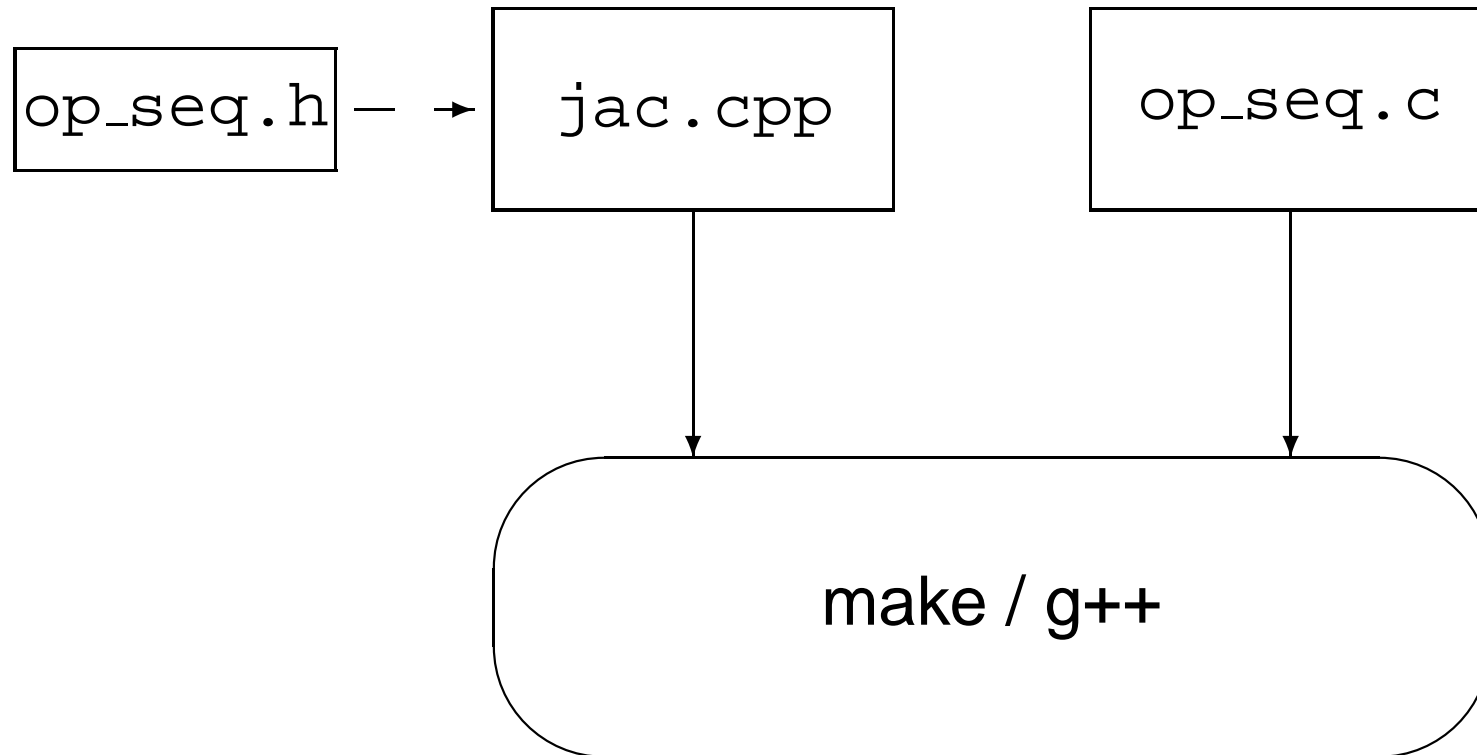
User build processes

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

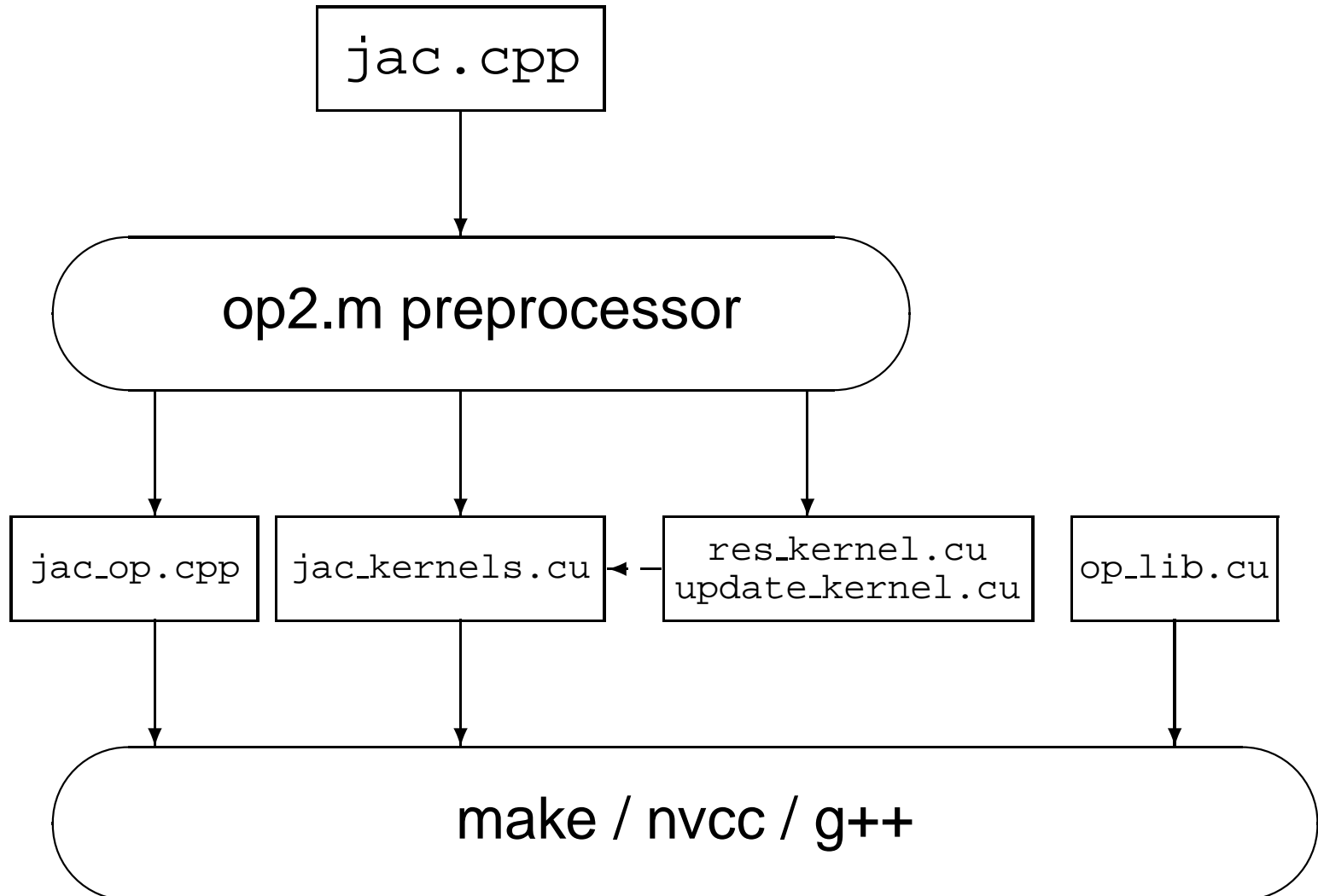
Sequential build process

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

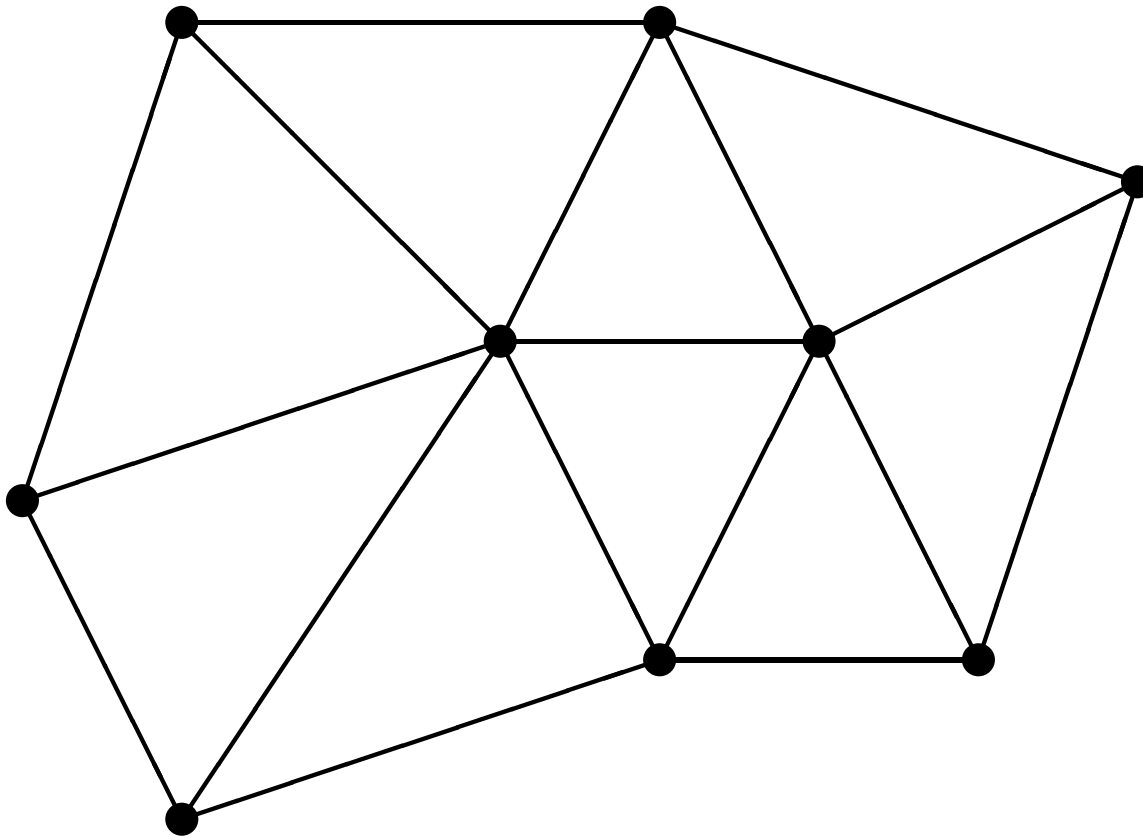
One important difference from MPI parallelisation

- 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.

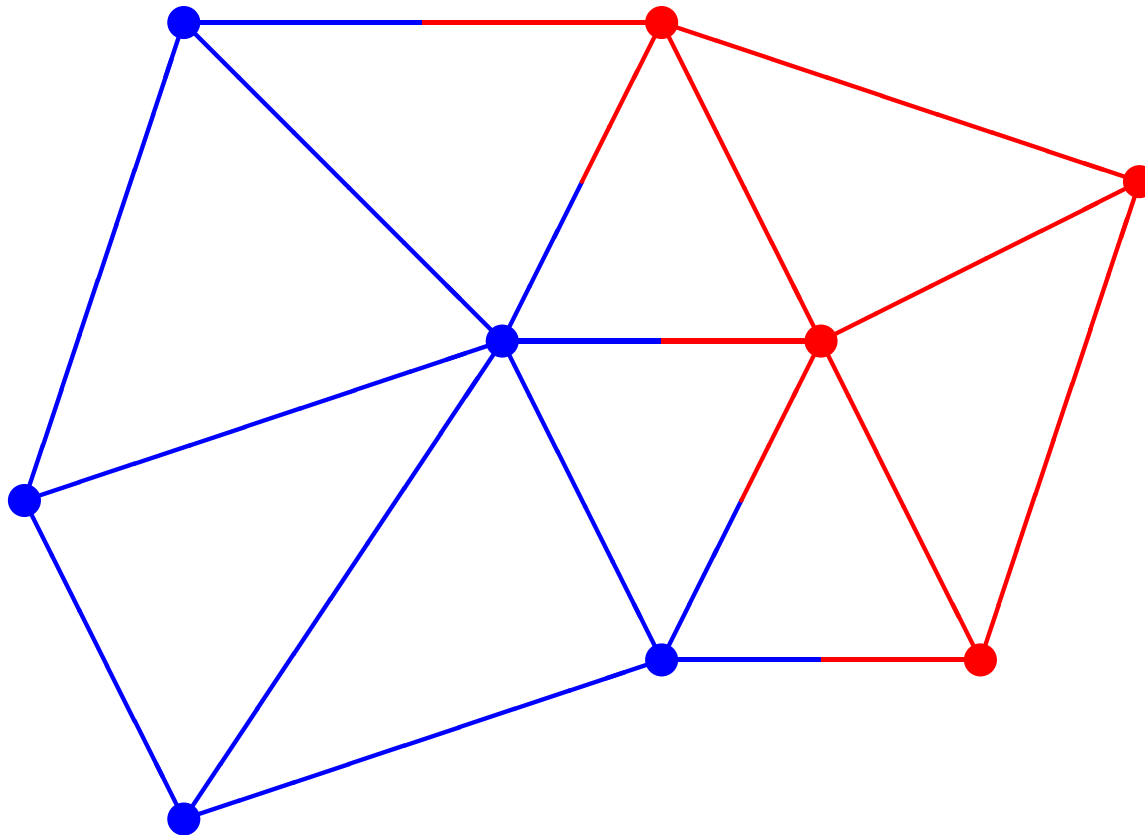
e.g. potential problem when two edges update same node



Data dependencies

Method 1: “owner” of nodal data does edge computation

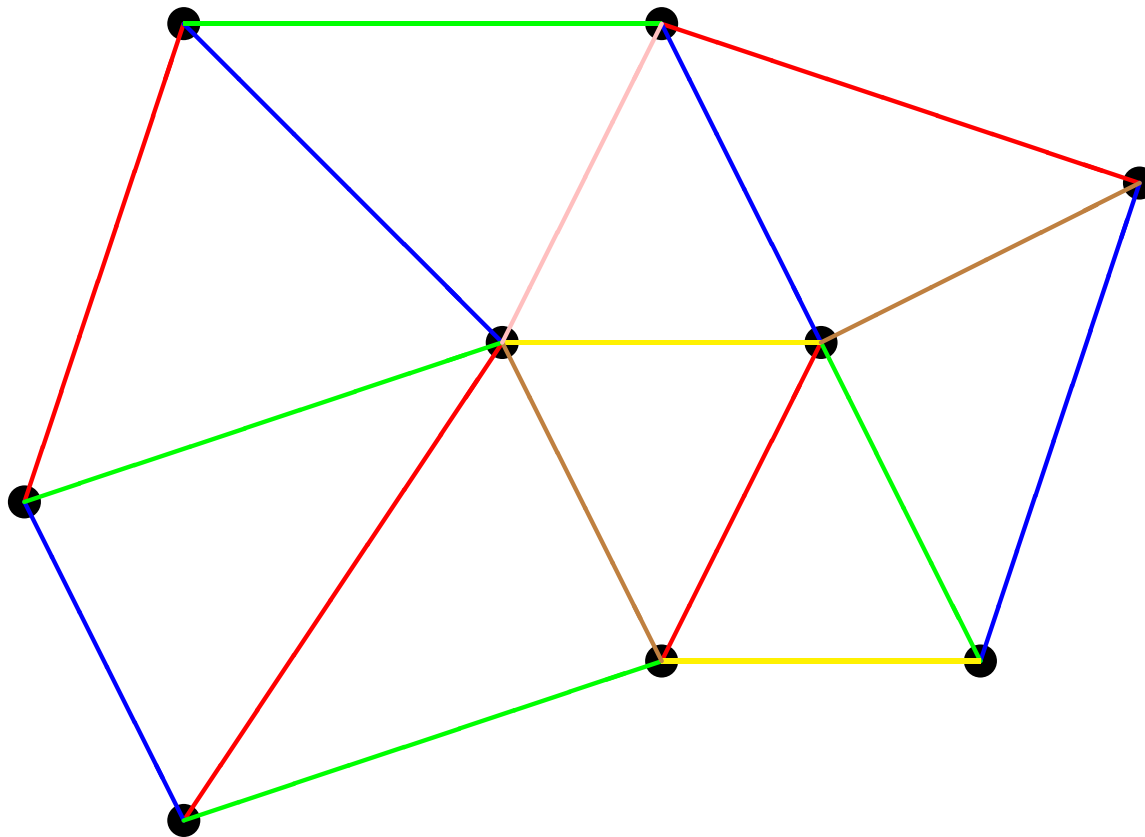
- drawback is redundant computation when the two nodes have different “owners”



Data dependencies

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

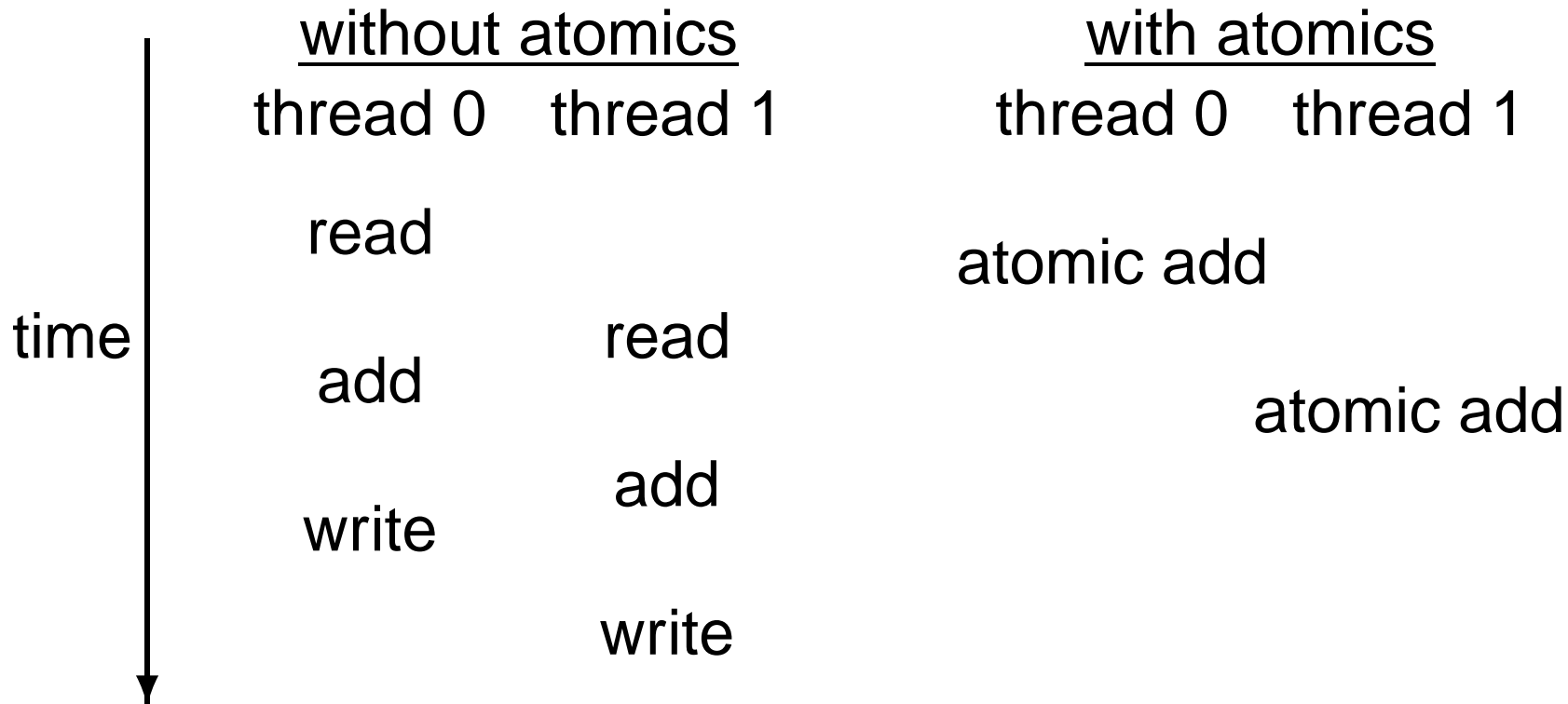
- parallel execution for each color, then synchronize
- possible loss of data reuse and some parallelism



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



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

- initial prototype (with code parser/generator written in MATLAB!) can generate:
 - CUDA code for a single GPU
 - OpenMP code for multiple CPUs
- Imperial College have re-implemented CUDA generator in Rose
- airfoil test case shows:
 - $28\times$ speedup on a single GPU
 - $7\times$ speedup for 2 quad-core CPUsrelative to a single CPU thread
- new postdoc will work on MPI implementation, using Parmetis or Chaco for domain decomposition

Airfoil test code

- 2D Euler equations, cell-centred finite volume with scalar dissipation (minimal compute per memory reference – should consider switching to characteristic smoothing)
- roughly 1.5M edges, 0.75M cells
- 5 parallel loops:
 - `save_soln` (direct over cells)
 - `adt_calc` (indirect over cells)
 - `res_calc` (indirect over edges)
 - `bres_calc` (indirect over boundary edges)
 - `update` (direct over cells with RMS reduction)

Airfoil test code

- factor 2-4 data reuse in indirect access, but cache efficiency not known (need extra coding for this, or hardware monitoring)
- some routines seem close to bandwidth-limited; all have at least 30% bandwidth utilisation
- only factor 7 speedup on 16 CPU threads (2×4 cores, but hyperthreaded) – we think due to memory bandwidth limits
- single precision CUDA limited to 32 registers; double precision needs up to 63 (but no spillage into “local” memory)

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 maths 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) 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 maybe 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

3) 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 working with me on a 3rd year project on this

Lessons learned so far

4) Unstructured grids lead to lots of integer pointer arithmetic

- “free” on CPUs due to integer pipelines
- costs almost as much as floating point operations on GPU, at least in single precision
- reduces maximum benefits from GPUs?

5) Open source development leads to great collaboration

- others test code and find bugs – even better, they figure out how to fix them
- will share code development in the future
- project webpage at <http://people.maths.ox.ac.uk/gilesm/op2/>

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, Yoon Ho, David Radford (Rolls-Royce)
- Tom Bradley, Jon Cohen and others (NVIDIA)
- EPSRC, NVIDIA and Rolls-Royce for financial support
- Oxford Supercomputing Centre

Liszt: discussion points?

- different parallel strategies (MPI-level and GPU-level)
- hybrid approach at GPU-level (re-order threads by color to reduce warp divergence)
- performance assessment – which tools are best for identifying bottlenecks?
- Joe performance analysis – bandwidth-limited or compute-bound?
- comparative assessment, Liszt vs. OP2, using Joe and Airfoil test codes?

Liszt: discussion points?

- lessons learned from multi-GPU computation? is communication overlapped with computation?
- register usage – concerns? remedies?
- pros and cons of Rose vs. Scala (but I have very limited understanding in this area)
- any interest in auto-tuning?
- any plans for automatic mesh refinement? does this spoil the purity of the programming model?
- AVX code generation? experience with vectorisation using Intel's `icc`?

NVIDIA: discussion points?

- register usage
 - how to minimize (use of shared memory)
 - compiler optimizations
 - spillage to L1 cache
- assessing cache misses
- future bandwidth developments
- various CUDA questions