• Complexity of Unstructured Mesh Applications

• The OP2 Library

• Efficient Support to Multiple Architectural Back-Ends

• Experiments
Unstructured Meshes

- The mesh can be composed of triangles, squares, etc...
  - With different sizes
  - With a different number of neighbours
- Refinement is applied when a higher detail level is needed
- In general terms, a larger mesh involves more computation than a smaller mesh
Structured Vs Unstructured Mesh Applications

- In **structured mesh** applications (stencil programs):
  - There is a formula to describe how elements are connected between themselves (e.g. $i-1, i+1$)
  - Optimisations can be introduced at compile-time by looking at the formula (plenty of books and thesis about this subject...)

- In **unstructured mesh** applications
  - The same program can be applied to different meshes
  - Connections are only known when the file describing the mesh is loaded at the beginning of the execution
  - Optimisations can’t be performed at compile-time, but they are moved at run-time
  - Therefore, the optimisation cost cannot be assumed to be negligible
  - Less research results are available in this field
Architectural Back-Ends and Optimisations

• Currently, production-level unstructured mesh applications are executed on large clusters of multicores
  - For days or weeks, depending on the size of the mesh and the kind of computation
• How can we accelerate this?
  - **Graphics Processing Units** (GPUs)
  - **SIMD support at assembler level** (AVX/SSE)
  - **FPGAs**?
• Every back-end type features different optimisations
• Also every technology has its own optimisations, e.g. Nvidia and AMD GPUs
The OP2 Approach

• The issue of programming (and optimising programs for) different back-end architectures is huge:
  - The case of unstructured mesh applications makes the problem even worse
  - How can we attack it?

• Narrow the supported applications
  - In OP2 the order of iterations of (parallel) loops must not affect the final result
  - OP2 only supports implicit methods, not explicit ones
  - Examples: pseudo-time-stepping meth., multigrid meth., etc..
  - Cannot handle cases like the finite element method

• Optimise the OP2 implementation
  - Using an active library approach, in which we generate different back-end code transforming the input program (i.e. source-to-source)
  - Support different optimised run-time library for each back-end
Fig. 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets: nodes and cells. There are 16 nodes and 9 cells which can be defined using the OP2 API as shown in Fig. 2.

In our previous works [10, 11] we have detailed the OP2 API for code development in C/C++. Here we will introduce the Fortran API.

```
integer :: numNodes = 16
integer :: numCells = 9

type :: nodes, cells
integer :: numNodes = 16
integer :: numCells = 9

integer, dimension(36) :: cell_map = 
& (/ 0,1,5,4, 1,2,6,5, 2,3,7,6, & 
& 4,5,9,8,5,6,10,9,6,7,11,10, &
& 8,9,13,12,9,10,14,13,10,11,15,14 /)

type :: cellsToNodes

call op_decl_set ( numNodes, nodes )
call op_decl_set ( numCells, cells )
call op_decl_map ( cells, nodes, 4, &
& cell_map, cellsToNodes )
```

### Mesh declaration

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

---

**The OP2 User Interface**

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets
Fig. 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets: nodes and cells. There are 16 nodes and 9 cells, which can be defined using the OP2 API as shown in Fig. 2. In our previous works [10, 11] we have detailed the OP2 API for code development in C/C++. Here we will introduce the Fortran API.

**Mesh declaration**

```fortran
integer(4) :: numNodes = 16
integer(4) :: numCells = 9

type(op_set) :: nodes, cells

integer(4), dimension(36) :: cell_map = &
& (/ 0,1,5,4, 1,2,6,5, 2,3,7,6, &
& 4,5,9,8,5,6,10,9,6,7,11,10, &
& 8,9,13,12,9,10,14,13,10,11,15,14 /)

type(op_map) :: cellsToNodes

call op_decl_set (numNodes, nodes)
call op_decl_set (numCells, cells)
call op_decl_map (cells, nodes, 4, &
& cell_map, cellsToNodes)
```

**The OP2 User Interface**

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets
Fig. 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets: nodes and cells. There are 16 nodes and 9 cells which can be defined using the OP2 API as shown in Fig 2. In our previous works [10, 11] we have detailed the OP2 API for code development in C/C++. Here we will introduce the Fortran API.

```fortran
integer :: numNodes = 16
integer :: numCells = 9

type :: nodes, cells

integer, dimension(36) :: cell_map = (/ 0, 1, 5, 4, 1, 2, 6, 5, 2, 3, 7, 6, & 4, 5, 9, 8, 5, 6, 10, 9, 6, 7, 11, 10, & 8, 9, 13, 12, 9, 10, 14, 13, 10, 11, 15, 14 /)

type :: cellsToNodes

call op_decl_set ( numNodes, nodes )
call op_decl_set ( numCells, cells )
call op_decl_map ( cells, nodes, 4, &
& cell_map, cellsToNodes )
```

The OP2 User Interface

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Mesh declaration
Fig. 1. An example mesh with node and quadrilateral cell indices and associated data values in parenthesis.

Fig. 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets: nodes and cells. There are 16 nodes and 9 cells which can be defined using the OP2 API as shown in Fig. 2.

In our previous works [10, 11] we have detailed the OP2 API for code development in C/C++. Here we will introduce the Fortran API.

```
integer :: numNodes = 16
integer :: numCells = 9

type :: set

integer dimension :: cell_map = (/ 0,1,5,4, 1,2,6,5, 2,3,7,6, & 4,5,9,8,5,6,10,9,6,7,11,10, & 8,9,13,12,9,10,14,13,10,11,15,14 /)

type :: map

call op_decl_set ( numNodes, nodes )
call op_decl_set ( numCells, cells )
call op_decl_map ( cells, nodes, 4, &
  cell_map, cellsToNodes )
```

Mesh declaration

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

The OP2 User Interface

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets
The OP2 User Interface

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Mesh declaration

```fortran
integer(4) :: numNodes = 16
integer(4) :: numCells = 9

type(op_set) :: nodes, cells

integer(4), dimension(36) :: cell_map = &
  / 0,1,5,4, 1,2,6,5, 2,3,7,6, &
  & 4,5,9,8,5,6,10,9,6,7,11,10, &
  & 8,9,13,12,9,10,14,13,10,11,15,14 /

type(op_map) :: cellsToNodes

call op_decl_set ( numNodes, nodes )
call op_decl_set ( numCells, cells )
call op_decl_map ( cells, nodes, 4, &
  & cell_map, cellsToNodes )
```

Further restriction: the number of mapped elements must be the same for all elements in a set.
• Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Data declaration

```plaintext
real(8), dimension(9) :: cell_data = &
  & (/ 0.128, 0.345, 0.224, 0.118, &
  & 0.246, 0.324, 0.112, 0.928, &
  & 0.237 /)

real(8), dimension(9) :: cell_data_updated

real(8), dimension(16) :: nodes_data = &
  & (/ 5.3, 6.8, 7.8, 5.4, 2.6, 3.6, &
  & 7.5, 6.2, 1.8, 3.9, 2.5, 6.6, &
  & 1.3, 2.8, 3.9, 8.8 /)

type(op_dat) :: dataCells, dataCellsUpdated, dataNodes

call op_decl_dat ( cells, 1, cell_data, dataCells )

call op_decl_dat ( cells, 1, cell_data_updated, &
  & dataCellsUpdated )

call op_decl_dat ( nodes, 1, nodes_data, dataNodes )
```

Fig. 3. Example of data array declaration and OP2 variables.
The OP2 User Interface (cont.)

• Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Data declaration

```fortran
real(8), dimension(9) :: cell_data = &
  / 0.128, 0.345, 0.224, 0.118, & & 0.246, 0.324, 0.112, 0.928, & & 0.237 /

real(8), dimension(9) :: cell_data_updated

real(8), dimension(16) :: nodes_data = &
  / 5.3, 6.8, 7.8, 5.4, 2.6, 3.6, & & 7.5, 6.2, 1.8, 3.9, 2.5, 6.6, & & 1.3, 2.8, 3.9, 8.8 /

type(op_dat) :: dataCells, dataCellsUpdated, dataNodes

call op_decl_dat ( cells, 1, cell_data, dataCells )

call op_decl_dat ( cells, 1, cell_data_updated, & & dataCellsUpdated )

call op_decl_dat ( nodes, 1, nodes_data, dataNodes )
```

Fig. 3. Example of data array declaration and OP2 variables.
The OP2 User Interface (cont.)

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Data declaration

```plaintext
real(8), dimension(9) :: cell_data = &
                          & (/ 0.128, 0.345, 0.224, 0.118, &
                           & 0.246, 0.324, 0.112, 0.928, &
                           & 0.237 /)

real(8), dimension(9) :: cell_data_updated

real(8), dimension(16) :: nodes_data = &
                         & (/ 5.3, 6.8, 7.8, 5.4, 2.6, 3.6, &
                          & 7.5, 6.2, 1.8, 3.9, 2.5, 6.6, &
                          & 1.3, 2.8, 3.9, 8.8 /)

type(op_dat) :: dataCells, dataCellsUpdated, dataNodes
call op_decl_dat ( cells, 1, cell_data , dataCells )
call op_decl_dat ( cells, 1, cell_data_updated, &
                 & dataCellsUpdated )
call op_decl_dat ( nodes, 1, nodes_data, dataNodes )
```

Fig. 3. Example of data array declaration and OP2 variables.
The OP2 User Interface (cont.)

- Calls to:
  - Declare sets, their connections, and associated data
  - Perform computations over sets

Data declaration

```fortran
real(8), dimension(9) :: cell_data = &
  (/ 0.128, 0.345, 0.224, 0.118, &
  0.246, 0.324, 0.112, 0.928, &
  0.237 /)

real(8), dimension(9) :: cell_data_updated

real(8), dimension(16) :: nodes_data = &
  (/ 5.3, 6.8, 7.8, 5.4, 2.6, 3.6, &
  7.5, 6.2, 1.8, 3.9, 2.5, 6.6, &
  1.3, 2.8, 3.9, 8.8 /)

type(op_dat) :: dataCells, dataCellsUpdated, dataNodes

call op_decl_dat ( cells, 1, cell_data, dataCells )
call op_decl_dat ( cells, 1, cell_data_updated, &
  dataCellsUpdated )
call op_decl_dat ( nodes, 1, nodes_data, dataNodes )
```

Fig. 3. Example of data array declaration and OP2 variables.
In OP2 we can express the application of a certain subroutine/function to all elements of a given set.

We have also to tell on which data the subroutine operates.
OP2 Parallelism

- In OP2 we can express the application of a certain subroutine/function to all elements of a given set.
- We have also to tell on which data the subroutine operates.
OP2 Parallelism

- In OP2 we can express the application of a certain subroutine/function to all elements of a given set.
- We have also to tell on which data the subroutine operates.

The kernel is expressed for a generic cell:

```fortran
subroutine kernel ( cellUpdated, cell, node1, node2, node3, node4 )
  real(8), dimension(1) :: cellUpdated
  real(8), dimension(1) :: cell
  real(8), dimension(1) :: node1
  real(8), dimension(1) :: node2
  real(8), dimension(1) :: node3
  real(8), dimension(1) :: node4

  cellUpdated(1) = cell(1) + node1(1) + node2(1) + node3(1) + node4(1)
end subroutine kernel
```
In OP2 we can express the application of a certain subroutine/function to all elements of a given set.

We have also to tell on which data the subroutine operates.

The parallel loop is about all elements in the set.

```plaintext
subroutine kernel ( cellUpdated, cell, node1, node2, node3, node4 )
  real(8), dimension(1) :: cellUpdated
  real(8), dimension(1) :: cell
  real(8), dimension(1) :: node1
  real(8), dimension(1) :: node2
  real(8), dimension(1) :: node3
  real(8), dimension(1) :: node4

  cellUpdated(1) = cell(1) + node1(1) + node2(1) + node3(1) + node4(1)
end subroutine kernel
```

```plaintext
call op_par_loop ( kernel, cells, &
  & dataCellsUpdated, -1, OP_ID, OP_WRITE,
  & dataCells, -1, OP_ID, OP_READ,
  & dataNodes, 1, cellsToNodes, OP_READ,
  & dataNodes, 2, cellsToNodes, OP_READ,
  & dataNodes, 3, cellsToNodes, OP_READ,
  & dataNodes, 4, cellsToNodes, OP_READ )
```
• In OP2 we can express the application of a certain subroutine/function to all elements of a given set.

• We have also to tell on which data the subroutine operates.

The parallel loop is about all elements in the set.

```fortran
call op_par_loop ( kernel, cells &
    & dataCellsUpdated, -1, OP_ID, OP_WRITE, &
    & dataCells, -1, OP_ID, OP_READ, &
    & dataNodes, 1, cellsToNodes, OP_READ, &
    & dataNodes, 2, cellsToNodes, OP_READ, &
    & dataNodes, 3, cellsToNodes, OP_READ, &
    & dataNodes, 4, cellsToNodes, OP_READ )

subroutine kernel ( cellUpdated, cell, node1, node2, node3, node4 )
    real(8), dimension(1) :: cellUpdated
    real(8), dimension(1) :: cell
    real(8), dimension(1) :: node1
    real(8), dimension(1) :: node2
    real(8), dimension(1) :: node3
    real(8), dimension(1) :: node4

    cellUpdated(1) = cell(1) + node1(1) + node2(1) + node3(1) + node4(1)
end subroutine kernel
```
What does the compiler do?

```c
// OP initialisation
op_init(argc, argv);

op_set nodes(nnode, NULL);
op_set edges(nedge, NULL);
op_set cells(ncell, NULL);

op_ptr pedge(edges, nodes, 2, edge);
op_ptr pcell(edges, cells, 2, ecell);
op_ptr pcell(cells, nodes, 4, cell);

op_dat<int> p_boun(edges, 1, boun);
op_dat<float> p_x(nodes, 2, x);
op_dat<float> p_q(cells, 4, q);
op_dat<float> p_qold(cells, 4, qold);
op_dat<float> p_adt(cells, 1, adt);
op_dat<float> p_res(cells, 4, res);

op_decl_const(1, &gam);
op_decl_const(1, &gml);
op_decl_const(1, &cfl);
op_decl_const(1, &eps);
op_decl_const(1, &mach);
op_decl_const(1, &alpha);
op_decl_const(1, &air_const);

op_diagnostic_output();

// main time-marching loop
niter = 1000;
float saveTime = 0.0f, adtTime = 0.0f, resTime = 0.0f, updateTime = 0.0f;
for(int iter=1; iter<niter; iter++) {

    save old flow solution
    saveTime += op_par_loop_2(save_soln, cells,
        &p_q, 0, NULL, OP_READ,
        &p_qold, 0, NULL, OP_WRITE);

    // predictor/corrector update loop
    for(int k=0; k<2; k++) {

        calculate area/timstep
        adtTime += op_par_loop_6(adt_calc, cells,
            &p_x, 0, &pcell, OP_READ,
            &p_x, 1, &pcell, OP_READ,
            &p_x, 2, &pcell, OP_READ,
```
What does the compiler do?

```c
// OP initialisation
op_init(argc, argv);

op_set nodes(nnode, NULL);
op_set edges(nedge, NULL);
op_set cells(ncell, NULL);

op_ptr pedge(edges, nodes, 2, edge);
op_ptr pcell(edges, cells, 2, ecell);
op_ptr pcells(edges, cells, 4, cell);

op_dat<int> p_boun(edges, 1, boun);
op_dat<float> p_x(nodes, 2, x);
op_dat<int> p_q(cells, 4, q);
op_dat<float> p_qold(cells, 4, qold);
op_dat<float> p_adt(cells, 1, adt);

op_decl_const(1, &gam);
op_decl_const(1, &gm1);
op_decl_const(1, &cfl);
op_decl_const(1, &eps);
op_decl_const(1, &mach);
op_decl_const(1, &alpha);
op_decl_const(1, &air_const);

op_diagnostic_output();

// main time-marching loop
niter = 1000;
float saveTime = 0.0f, adtTime = 0.0f, resTime = 0.0f, updateTime = 0.0f;
for(int iter=1; iter<=niter; iter++) {
    // save old flow solution
    saveTime += op_par_loop_2(save_soln, cells,
        &p_q, 0, NULL, OP_READ,
        &p_qold, 0, NULL, OP_WRITE);

    // predictor/corrector update loop
    for(int k=0; k<2; k++) {
        // Calculate area/time step
        adtTime += op_par_loop_6(adt_calc, cells,
            &p_x, 0, &pcell, OP_READ,
            &p_x, 1, &pcell, OP_READ,
            &p_x, 2, &pcell, OP_READ,
            &p_x, 3, &pcell, OP_READ,
            &p_x, 4, &pcell, OP_READ,
            &p_x, 5, &pcell, OP_READ,
            &p_x, 6, &pcell, OP_READ);
```
OP2 Compiler

What does the compiler do?

// OP initialisation
op_init(argc, argv);

// OP set
op_set nodes(nnode, NULL);
op_set edges(nedge, NULL);
op_set_cells(ncell, NULL);

// OP ptr
op_ptr pedge(edges, nodes, 2, edge);
op_ptr pecell(edges, cells, 2, ecell);
op_ptr pcell(cells, nodes, 4, cell);

// OP data
op_dat<int> p_boun(edges, 1, boun);
op_dat<float> p_x(nodes, 2, x);
op_dat<float> p_q(cells, 4, q);
op_dat<float> p_qold(cells, 4, qold);
op_dat<float> p_adt(cells, 1, adt);
op_dat<float> p_res(cells, 4, res);

// OP decl const
op_decl_const(1, &gam);
op_decl_const(1, &gm1);
op_decl_const(1, &cfl);
op_decl_const(1, &eps);
op_decl_const(1, &mach);
op_decl_const(1, &alpha);
op_decl_const(1, &air_const);

// OP diagnostic output
op_diagnostic_output();

// main time-marching loop
niter = 1000;
float saveTime = 0.0f, adtTime = 0.0f, resTime = 0.0f, updateTime = 0.0f;
for(int iter=1; iter<=niter; iter++) {

// save old flow solution
saveTime += op_par_loop_2(save_soln, cells,
&p_q, 0, NULL, OP_READ,
&p_qold, 0, NULL, OP_WRITE);

// predictor/corrector update loop
for(int k=0; k<2; k++)

// Calculate area/timestep
adtTime += op_par_loop_6(adt_calc, cells,
&p_x, 0, pcell, OP_READ,
&p_x, 1, pcell, OP_READ,
&p_x, 2, pcell, OP_READ,

// CUDA kernel function
__global__ void op_cuda_adt_calc(
    float *ind_arg0, int *ind_arg0_maps,
    short *arg0_maps,
    // set shared memory pointers
    int nbyte = 0;
    ind_arg0_s = (float *) &shared[nbyte];
    }
}

// get
#if defined
__syncthreads();
#else
if (0
__syncthreads();
#endif

// process set elements
for (int n=nthreadIdx.x; n<ind_arg0_size*2; n+=blockDim.x)
    ind_arg0_s[n] = ind_arg0_addr[n/2];

// user-supplied kernel call
adt_calc(ind_arg0_s[n], n, ncell, n=blockDim.x)

Monday, 29 August 2011
OP2 Compiler

What does the compiler do?

```c
// OP initialisation
op_init(argc,argv);

op_set nodes(nnode, NULL);
op_set edges(nedge, NULL);
op_set cells(ncell, NULL);

op_ptr pedge(edges, nodes, NULL);
op_ptr pecell(edges, cells, NULL);

op_ptr pcell(cells, nodes, NULL);

op_ptr pedge(edges, nodes, 2, NULL);

op_ptr pecell(edges, cells, 2, NULL);

op_ptr pcell(cells, nodes, 4, NULL);

op_data<int> p_boun(edges, 1, NULL);

op_data<float> p_x(nodes, 2, NULL);

op_data<float> p_q(cells, 4, NULL);

op_data<float> p_qold(cells, 4, NULL);

op_data<float> p_adt(cells, 1, NULL);

op_data<float> p_res(cells, 4, NULL);

op_decl_const(1, &gam);

op_decl_const(1, &gm1);

op_decl_const(1, &cfl);

op_decl_const(1, &eps);

op_decl_const(1, &mach);

op_decl_const(1, &alpha);

op_decl_const(1, &air_const);

op_diagnostic_output();

// main time-marching loop
niter = 1000;
float saveTime = 0.0f, adtTime = 0.0f, resTime = 0.0f, updateTime = 0.0f;
for(int iter=1; iter<niter; iter++) {
    // save old flow solution
    saveTime += op_par_loop_2(save_soln, cells,
        &p_q, 0, NULL, OP_READ,
        &p_qold, 0, NULL, OP_WRITE);

    // predictor/corrector update loop
    for(int k=0; k<2; k++) {
        // calculate area/timstep
        adtTime += op_par_loop_6(adt_calc, cells,
            &p_x, 0, pcell, OP_READ,
            &p_x, 1, pcell, OP_READ,
            &p_x, 2, pcell, OP_READ,
            &p_x, 3, pcell, OP_READ,
            &p_x, 4, pcell, OP_READ,
            &p_x, 5, pcell, OP_READ,
            &p_x, 6, pcell, OP_READ,
            &p_x, 7, pcell, OP_READ,
            &p_x, 8, pcell, OP_READ,
            &p_x, 9, pcell, OP_READ,
            &p_x, 10, pcell, OP_READ,
            &p_x, 11, pcell, OP_READ,
            &p_x, 12, pcell, OP_READ,
            &p_x, 13, pcell, OP_READ,
            &p_x, 14, pcell, OP_READ,
            &p_x, 15, pcell, OP_READ,
            &p_x, 16, pcell, OP_READ,
            &p_x, 17, pcell, OP_READ,
            &p_x, 18, pcell, OP_READ,
            &p_x, 19, pcell, OP_READ,
            &p_x, 20, pcell, OP_READ,
            &p_x, 21, pcell, OP_READ,
            &p_x, 22, pcell, OP_READ,
            &p_x, 23, pcell, OP_READ,
            &p_x, 24, pcell, OP_READ,
            &p_x, 25, pcell, OP_READ,
            &p_x, 26, pcell, OP_READ,
            &p_x, 27, pcell, OP_READ,
            &p_x, 28, pcell, OP_READ,
            &p_x, 29, pcell, OP_READ,
            &p_x, 30, pcell, OP_READ,
            &p_x, 31, pcell, OP_READ,
            &p_x, 32, pcell, OP_READ,
            &p_x, 33, pcell, OP_READ,
            &p_x, 34, pcell, OP_READ,
            &p_x, 35, pcell, OP_READ,
            &p_x, 36, pcell, OP_READ,
            &p_x, 37, pcell, OP_READ,
            &p_x, 38, pcell, OP_READ,
            &p_x, 39, pcell, OP_READ,
            &p_x, 40, pcell, OP_READ,
            &p_x, 41, pcell, OP_READ,
            &p_x, 42, pcell, OP_READ,
            &p_x, 43, pcell, OP_READ,
            &p_x, 44, pcell, OP_READ,
            &p_x, 45, pcell, OP_READ,
            &p_x, 46, pcell, OP_READ,
            &p_x, 47, pcell, OP_READ,
            &p_x, 48, pcell, OP_READ,
            &p_x, 49, pcell, OP_READ,
            &p_x, 50, pcell, OP_READ,
            &p_x, 51, pcell, OP_READ,
            &p_x, 52, pcell, OP_READ,
            &p_x, 53, pcell, OP_READ,
            &p_x, 54, pcell, OP_READ,
            &p_x, 55, pcell, OP_READ,
            &p_x, 56, pcell, OP_READ,
            &p_x, 57, pcell, OP_READ,
            &p_x, 58, pcell, OP_READ,
            &p_x, 59, pcell, OP_READ,
            &p_x, 60, pcell, OP_READ,
            &p_x, 61, pcell, OP_READ,
            &p_x, 62, pcell, OP_READ,
            &p_x, 63, pcell, OP_READ,
            &p_x, 64, pcell, OP_READ,
            &p_x, 65, pcell, OP_READ,
            &p_x, 66, pcell, OP_READ,
            &p_x, 67, pcell, OP_READ,
            &p_x, 68, pcell, OP_READ,
            &p_x, 69, pcell, OP_READ,
            &p_x, 70, pcell, OP_READ,
            &p_x, 71, pcell, OP_READ,
            &p_x, 72, pcell, OP_READ,
            &p_x, 73, pcell, OP_READ,
            &p_x, 74, pcell, OP_READ,
            &p_x, 75, pcell, OP_READ,
            &p_x, 76, pcell, OP_READ,
            &p_x, 77, pcell, OP_READ,
            &p_x, 78, pcell, OP_READ,
            &p_x, 79, pcell, OP_READ,
            &p_x, 80, pcell, OP_READ,
            &p_x, 81, pcell, OP_READ,
            &p_x, 82, pcell, OP_READ,
            &p_x, 83, pcell, OP_READ,
            &p_x, 84, pcell, OP_READ,
            &p_x, 85, pcell, OP_READ,
            &p_x, 86, pcell, OP_READ,
            &p_x, 87, pcell, OP_READ,
            &p_x, 88, pcell, OP_READ,
The Big Picture

- C/C++ OP2
- Fortran OP2

OP2 Compiler
- C/C++ Parser
- Fortran Parser

Loop Optimisations (fission, fusion)

AST Transformation
- OpenMP
- CUDA
- OpenCL
- AVX/SSE
- MPI

- Fortran Unparser
- C/C++ Unparser

OP2 Run-Time
- (partitioning, colouring)

- GPGPUs
- AVX
- GPGPU Clusters
- Multicore Clusters
- FPGAs

Monday, 29 August 2011
Run-Time Support

- The OP2 run-time support is a critical piece of code:
  - it solves race condition issues
  - it optimises loop execution
Run-Time Support

- The OP2 run-time support is a critical piece of code:
  - it solves race condition issues
  - it optimises loop execution

Race Conditions (shared memory):
- a parallel loop iterates over faces
- it increments the values on nodes (i.e. reads the node value and increments it)
- What if we run in parallel two iterations?
The OP2 run-time support is a critical piece of code:

- it solves race condition issues
- it optimises loop execution

Race Conditions (shared memory):

- a parallel loop iterates over faces
- it increments the values on nodes (i.e. reads the node value and increments it)

What if we run in parallel two iterations?

Solution 1: locks on nodes data

Solution 2: colour faces and execute colour by colour
Run-Time Support

• The OP2 run-time support is a critical piece of code:
  - it solves race condition issues
  - it optimises loop execution

• Race Conditions (shared memory):
  - a parallel loop iterates over faces
  - it increments the values on nodes (i.e. reads the node value and increments it)
  - What if we run in parallel two iterations?

• Solution 1: locks on nodes data
• Solution 2: colour faces and execute colour by colour

As locking mechanisms in GPUs are still too costly, the OP2 runtime for GPUs uses colouring
OP2 Run-Time Optimisations for GPUs
We partition the iteration set and we assign each partition to a different CUDA core.

We colour also partitions to avoid data conflicts.

We map data from device to shared memory: maximise data re-use, minimise memory traffic to device memory.
We partition the iteration set and we assign each partition to a different CUDA core.

We colour also partitions to avoid data conflicts.

We map data from device to shared memory: maximise data re-use, minimise memory traffic to device memory.
- We partition the iteration set and we assign each partition to a different CUDA core
- We colour also partitions to avoid data conflicts
- We map data from device to shared memory: maximise data re-use, minimise memory traffic to device memory
- We partition the iteration set and we assign each partition to a different CUDA core.
- We colour also partitions to avoid data conflicts.
- We map data from device to shared memory: maximise data re-use, minimise memory traffic to device memory.
OP2 Plans

- This support is encapsulated in a set of basic routines, the most important one is the execution plan generator (or plan function)
  - It is executed only once for each op_par_loop in the program
  - It is subject to continuous improvement: it is hence programmed in C99 and used by C/C++ and Fortran CUDA programs generated by the OP2 compiler

- This sharing comes with a cost
  - if we want to use variables allocated dynamically in C from a Fortran program, we need to convert the associated pointers from C to Fortran notation (c_f_pointer in Fortran 2003 iso_c_binding support)
  - For a current bug in the PGI CUDA Fortran compiler we need to do this every time we enter a parallel loop, for all variables generated by the plan function
  - Luckily, the overhead is negligible!
Experimental Setup

• To study performance optimisations for unstructured meshes we implemented a representative application (Airfoil)

• Airfoil is much simpler than target applications like RR’s Hydra, but it exposes some of the performance features of Hydra loops
  - we tested airfoil on a relatively small mesh including 720K nodes, 720K cells and about 1.5 million edges

• Tests are executed on
  - A Nvidia Fermi M2050 for HPC applications
  - A Nvidia GTX460 for graphical applications

• Parameters of execution are:
  - Mini-Partition size
  - Number of threads in a block (i.e. executed on the same CUDA core
Numerical Results

- Airfoil performance: double precision floating point, 2K iterations of main loops
- On Nvidia M2050

- Interestingly, the optimal parameters for Fortran and C are different
  - Fortran: mini-partition = 128, block-size = 128
  - C: mini-partition = 64, block-size = 64
Numerical Results

- Airfoil performance: double precision floating point, 2K iterations of main loops
- On Nvidia GTX 460
- In this case optimal parameters are the same
- Performance gracefully degrades on a slower GPU
Conclusion

• Programming and optimising unstructured mesh applications on new HPC architectures is a challenge
  - new optimisations are needed, where every architecture needs different kinds of optimisations

• Our approach based on narrowing the target set of applications to be studied has revealed interesting performance aspects for GPUs
  - they can be directly applied to large production-level applications, like Hydra developed by Rolls Royce Plc
  - they are a first attempt to efficiently program GPUs for unconventional applications
  - can be possibly generalised for other architectures (e.g. AVX/SSE)