The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations

István Z. Reguly, Gihan R. Mudalige, Mike Giles
Oxford e-Research Centre, University of Oxford
Dan Curran, Simon McIntosh-Smith
Department of Computer Science, University of Bristol

WOLFHPC 2014 Workshop at SC 14
New Orleans, November 17, 2014
Introduction

• Importance of Domain Specific approaches in HPC
  • Performance, Maintenance, Future Proofing
  • But then again, you already know this…

• Originally from CFD: the OP2 domain specific active library for unstructured meshes
  • Active Library
  • Rolls-Royce Hydra, VOLNA tsunami simulation
  • C, Fortran + a reluctance for maintaining compilers
Multi-Block Structured Grids

- Structured grids are popular due to their implicit connectivity
- Commonly used in CFD with finite difference and finite volume algorithms
- Realistic codes tend to use many blocks, different resolutions
  - Cloverleaf: Nuclear/Defence
  - ROTORSIM @ Bristol: helicopter rotors - sliding planes
  - SBLI @ Southampton: compressible Navier-Stokes
Designing an abstraction

Challenge: design an abstraction that:

- Covers a wide range of applications
- Intuitive to use
- Abstracts away parallelisation and data movement
- Still specific enough so that we can make aggressive platform-specific optimisations
The OPS Abstraction

- **Blocks**
  - A dimensionality, no size
  - Serves to group datasets together

  \[
  \text{ops\_block} = \text{ops\_decl\_block}(\text{dim}, \text{name});
  \]

- **Datasets on blocks**
  - With a given arity, type, size, optionally stride

  \[
  \text{ops\_dat} = \text{ops\_decl\_dat}(\text{block}, \text{arity}, \text{size}, \text{halo}, \ldots, \text{name});
  \]

- **Stencils**
  - Number of points, with relative coordinate offsets, optionally strides

  \[
  \text{ops\_stencil} = \text{ops\_decl\_stencil}(\text{dim}, \text{npoints}, \text{points}, \text{name});
  \]
Computations

- The description of computations follows the Access-Execute abstraction

- Loop over a given block, accessing a number of datasets with given stencils and type of access, executing a kernel function on each one

- Principal assumption: order of iteration through the grid doesn’t affect the results

```c
void calc(double *a, const double *b) {
    a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)] + b[OPS_ACC1(0,1)] +
                      b[OPS_ACC1(1,0)];
}
```

**User kernel**

**Iteration range**

```c
int range[4] = {12,50,12,50};
ops_par_loop(calc, block, 2, range,
             ops_arg_dat(a,S2D_0,”double”,OPS_WRITE),
             ops_arg_dat(b,S2D_1,”double”,OPS_READ));
```
Computations

- This definition decouples the **specification** of computations from their parallel **implementation**

- No assumption about data layout or movement

- Parallelism is implicit

- Easy to understand, maintain

- Enough information to organise execution & apply optimisations
The OPS Abstraction

- Multi-Block API
  - User specified halo
  - Exchange manually triggered
- In development
  - Multigrid API
  - Sliding planes
- Future
  - AMR, Multi-material
What the abstraction lets us do

• The library “owns” all the data
  • Access to it only through API calls
• Description of computations implicitly contain parallelism
• We can organise execution: parallelism & data movement
  • Code generation
  • Back-end
Code generation

- We parse the OPS API calls
- Contain all the information
- Generate parallel implementations for
  - Sequential, OpenMP, OpenACC
  - CUDA, OpenCL
- Callbacks to backend

```c
#define OPS_ACC0(j,i) j*xdim0+i
#define OPS_ACC1(j,i) j*xdim1+i

//user kernel
void calc(double *a, const double *b) {...}

void ops_par_loop_calc(int ndim, int range,
                       ops_arg arg0, ops_arg arg1) {
    //set up pointers and strides
double *p_a0 = (double*)ops_base_ptr(range, arg0);
double *p_a1 = (double*)ops_base_ptr(range, arg1);
xim0 = arg0.dat->size[0]; xim1 = arg1.dat->size[0];
    //do the computation
    for(int j = 0; j < range[3]-range[2]; j++) {
        for(int i = 0; i < range[1]-range[0]; i++) {
            calc(&p_a0[j*xdim0+i],&p_a1[j*xdim1+i]);
        }
    }
```
Code generation

- Checking consistency with declared stencils
- Adding `const`, `restrict`, and other keywords
- Deploying optimisations

- **OpenMP**
  - Explicit assignment of a block of rows to each thread
  - NUMA issues!

- **CUDA & OpenCL**
  - 1 grid point per thread
  - Use of non-coherent cache
  - Runtime compilation

- **OpenACC**
  - Nested loop with OpenACC pragmas (kernels/loop)
  - Currently links to CUDA backend, and uses `deviceptr()`
Build process

Structured Mesh Application → OPS Application (C/C++ API)

OPS Source-to-Source translator (Python)

Modified Platform Specific OPS Application → Platform Specific Optimized Application Files

Conventional Compiler + compiler flags (e.g. Icc, nvcc, pgcc) → Platform Specific Binary Executable

Mesh (hdf5) → Hardware

OPS Platform Specific Optimized Backend libraries

Single Node CUDA

Single Node OpenMP

Cluster MPI

Cluster MPI+CUDA
Backend logic

- We know:
  - iteration range
  - what data is accessed, how
  - stencils
Distributed Memory

- How much halo for each dataset
- What exactly is modified
- On-demand messaging with aggregation
- Dirtybits to keep track of changes
Checkpointing

• On the granularity of parallel loops
  • We know exactly what data is accessed and how
  • We know when data leaves the realm of OPS
  • Need to save anything that leaves
  • No need to save data that is going to be overwritten
  • Fast-forward: re-start and just not do any of the computations
Checkpointing

- Only a few datasets touched in any loop
- Checkpointing regions
- Decide what needs to be save over a number of loops
- Save to local & neighbouring SSD
Lazy execution

- OPS API expresses everything involved with computations
- We know when data leaves OPS (e.g. reduction)
- Loop chaining abstraction
  - We can carry out operations, optimisations that span several loops
  - Queue up a number of kernels, trigger execution when e.g. a reduction is encountered
- Implemented, works well - so what can we do with it?
MPI messaging

Default messaging strategy:

- On-demand
- Given loops $v_0$ and $v_1$
- Satisfy all dependencies before executing $v_1$
MPI messaging I.

Strategy 1:

- Given dependencies between $v_0 \rightarrow v_1$ and $v_0 \rightarrow v_2$
- Combine messages to hide latency
MPI messaging II.

- Strategy 2
- Given dependency $v_1 \rightarrow v_3$ but none to $v_2$
- Hide latency of message
 MPI messaging III.

- Strategy 3
- Given dependencies between $v_0 \rightarrow v_1$ and $v_2 \rightarrow v_3$ but not $v_1 \rightarrow v_2$
MPI messaging III.

• Strategy 3

• Given dependencies between $v_0 \rightarrow v_1$ and $v_2 \rightarrow v_3$ but not $v_1 \rightarrow v_2$

• Exchange loops $v_1$ and $v_2$, and hide latency of messages
MPI Communication avoidance

Given a sequence of loops

• Iterate backwards through a loop chain and determine dependencies

• Exchange wider halo at the beginning of the chain
MPI Communication avoidance

Given a sequence of loops

- Iterate backwards through a loop chain and determine dependencies
- Exchange wider halo at the beginning of the chain
MPI Communication avoidance

Given a sequence of loops

- Iterate backwards through a loop chain and determine dependencies
- Exchange wider halo at the beginning of the chain
Given a sequence of loops

- Iterate backwards through a loop chain and determine dependencies
- Exchange wider halo at the beginning of the chain
MPI Communication avoidance

- Extend halo region
- Redundant computations
- Fewer communications points
  - Larger messages
  - Fewer datasets need exchange in the end
Cache blocking

- Similar idea to communication-avoiding algorithm, except not over MPI and not with redundant compute

- Cache blocking, tiling; lot of work out there on polyhedral compilers
Cache blocking

- Similar idea to communication-avoiding algorithm, except not over MPI and not with redundant compute

- Cache blocking, tiling; lot of work out there on polyhedral compilers
Cache blocking

- Similar idea to communication-avoiding algorithm, except not over MPI and not with redundant compute

- Cache blocking, tiling; lot of work out there on polyhedral compilers
CloverLeaf

- Mini app in the Mantevo suite
- 2D/3D Structured hydrodynamics
- Explicit compressible Euler
- \( \sim 6k \) LoC
- Existing parallelizations (OpenMP, MPI, OpenACC, CUDA, OpenCL)

- Porting effort & performance?
  - Re-engineering, readability, tools, debugging
  - Is it worth the effort - maintainability, performance?
Porting CloverLeaf

• Initial 2D version
  • Fortran to C, 85 loops
  • Took about 1-2 months to port (including development of OPS)
  • Debugging

• 3D version 5 days

• No more difficult than porting to e.g. CUDA, but you get one codebase
Performance - CPU

3840*3840 mesh, 87 iterations

Xeon E5-2680 @ 2.7 GHz
Intel 14.0, Intel MPI
Performance - GPU

3840*3840 mesh, 87 iterations

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>OpenCL</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPS</td>
<td>15.01</td>
<td>16.27</td>
<td>19.82</td>
</tr>
</tbody>
</table>

NVIDIA K20c, CUDA 6.0, PGI 14.2
Performance - Scaling

**Strong Scaling**
15360 x 15360 mesh (87 iterations)

Titan, Cray XK7

**Weak Scaling**
3840 x 3840 mesh per node (87 iterations)
Conclusions

• An abstraction for multi-block structured codes
• Covers a sufficiently wide range of applications
• Viability of the Active Library approach
  • Performance, Productivity, Maintainability
• Advanced optimisations relying on the access-execute and loop chaining abstractions

Thank you! istvan.reguly@oerc.ox.ac.uk