A framework for parallel unstructured grid applications on GPUs

Mike Giles
mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute
Oxford eResearch Centre

SIAM Conference on Parallel Processing for Scientific Computing 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
- conclusions
**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.
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

FE applications

FD/FV applications

mathematical abstraction (UFL)

computational abstraction (OP2)

CUDA

OpenCL

OpenMP/AVX

NVIDIA

AMD

Intel
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 generates 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_dat(op_set set, int dim, op_datatype type, T *dat, op_dat *data, char *name)

op_exit()
Parallel loop for user kernel with 3 arguments:

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

Example for sparse matrix-vector product:

```c
op_par_loop_3(res,"res", edges,
               p_A, -1,edges_id, 1,OP_FLOAT,OP_READ,
               p_u,  0,pedge2,  1,OP_FLOAT,OP_READ,
               p_du, 0,pedge1,  1,OP_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:

```
op_seq.h   →  jac.cpp
          /     \
          |       |
          |       |
          v       v
       make / g++
```

```
op_seq.c
```
CUDA build process

Preprocessor parses user code and generates new code:

```
jac.cpp
```

```
op2.m preprocessor
```

```
jac_op.cpp  res_kernel.cu  update_kernel.cu  op_lib.cu
```

```
make / nvcc / g++
```
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

<table>
<thead>
<tr>
<th></th>
<th>without atomics</th>
<th>with atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread 0</td>
<td>read</td>
<td>atomic add</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>thread 1</td>
<td></td>
<td>atomic add</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td></td>
</tr>
</tbody>
</table>
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
- plan to look at OpenCL and PGI FORTRAN CUDA
- waiting for new NVIDIA Fermi hardware to assess performance – expanded shared memory and L1/L2 caches will help a lot
- looking for collaborators, either as users or co-developers
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 (Imperial College)
- Jamil Appa, Pierre Moinier (BAE Systems), Leigh Lapworth (Rolls-Royce)
- Tom Bradley, Jonathan Cohen, Massimiliano Fatica, Patrick LeGresley, Gernot Ziegler (NVIDIA)
- EPSRC, NVIDIA and Rolls-Royce for financial support