OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures

Mike Giles, Gihan Mudalige, István Reguly
Carlo Bertolli, Paul Kelly

Oxford e-Research Centre / Imperial College

InPar 2012: Innovative Parallel Computing

May 13, 2012
Outline

- structured and unstructured grids
- software challenge
- user perspective (i.e. application developer)
  - API
  - build process
- implementation issues
  - hierarchical parallelism on GPUs
  - data dependency
  - code generation
  - auto-tuning
Structured and unstructured grids

Structured grids:
- logical \((i, j)\) indexing in 2D, \((i, j, k)\) in 3D, with implicit connectivity
- easy to parallelize, including on GPUs with L1/L2 caches

Unstructured grids:
- a collection of nodes, edges, etc., with explicit connectivity – e.g. mapping tables define connections from edges to nodes
- much harder to parallelize (not in concept so much as in practice) but a lot of existing literature on the subject
Software Challenge

- Application developers want the benefits of the latest hardware but are very worried about the development effort required.
- Want to exploit GPUs using CUDA, and CPUs using OpenMP/AVX.
- However, hardware is likely to change rapidly in the next few years, and developers can’t afford to keep changing their codes.

Solution?

- high-level abstraction to separate the user’s specification of the application from the details of the parallel implementation.
- Aim to achieve application level longevity together with near-optimal performance through re-targetting the back-end implementation.
open source project

based on OPlus (Oxford Parallel Library for Unstructured Solvers) developed over 10 years ago for industrial CFD code on distributed-memory clusters

supports application codes written in C++ or FORTRAN

looks like a conventional library, but uses code transformation to generate CUDA for NVIDIA GPUs and OpenMP/AVX for CPUs/MIC

keeps OPlus abstraction, but slightly modifies API
OP2 Abstraction

- sets (e.g. nodes, edges, faces)
- datasets (e.g. flow variables)
- mappings (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)

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 mappings (no dynamic grid adaptation)
OP2 API

void op_init(int argc, char **argv)

op_set op_decl_set(int size, char *name)

op_map op_decl_map(op_set from, op_set to,
                   int dim, int *imap, char *name)

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

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

void op_exit()
Example of parallel loop syntax for a sparse matrix-vector product:

```c
op_par_loop(res,"res", edges,
    op_arg_dat(A,-1,OP_ID,1,"float",OP_READ),
    op_arg_dat(u, 0,col,1,"float",OP_READ),
    op_arg_dat(du,0,row,1,"float",OP_INC));
```

This is equivalent to the C code:

```c
for (e=0; e<nedges; e++)
    du[row[e]] += A[e] * u[col[e]];
```

where each “edge” corresponds to a non-zero element in the matrix $A$, and $row$, $col$ give the corresponding row and column indices.
User build processes

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

- **sequential single-thread CPU execution**
  - no code generation – just uses a header file
  - purely for program development and debugging
- **CUDA (and OpenCL in the future) for single GPU**
- **OpenMP (and AVX in the future) for multicore CPU systems**
- **MPI plus any of the above for clusters**
CUDA build process

Preprocessor parses user code and generates new code:

```
jac.cpp
```

```
op2.m preprocessor
```

```
jac_op.cpp  jac_kernels.cu  res_kernel.cu
            update_kernel.cu
```

```
op_lib.cu
```

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

Mike Giles (OeRC)
Implementation Approach

The question now is how to deliver good performance on multiple GPUs

- **MPI distributed-memory parallelism (1-100)**
  - one MPI process for each GPU, with standard partitioning so that each partition fits within global memory of GPU
  - only halos need to be transferred from one GPU to another

- **block parallelism (100-2000)**
  - on each GPU, data is broken into mini-partitions, worked on separately and in parallel by different Streaming Multiprocessors within the GPU
  - each mini-partition is sized so that all of the indirect data can be held in shared memory and re-used as needed

- **thread parallelism (64-256)**
  - each mini-partition is worked on by a block of threads in parallel
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

MPI level: “owner” of nodal data does edge computation
- drawback is redundant computation when the two nodes have different “owners”
Data dependencies

Thread level: “color” edges so no two edges of the same color update the same node

- compute increments in parallel, then apply them color by color with synchronisation between
- similar strategy also used at thread block level to avoid race condition
Other implementation issues

- array-of-structs storage preferred to struct-of-arrays
  - better cache hits for indirect addressing
  - transfers between graphics memory and GPU still largely “coalesced”

- auto-tuning very useful to optimize size of partitions and number of threads
Airfoil test code

- 2D Euler equations, cell-centred finite volume method with scalar dissipation
- two test cases:
  - 1.5M edges, 0.75M cells
  - 15M edges, 7.5M 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

Library is instrumented to give lots of diagnostic info:

new execution plan #1 for kernel res_calc
number of blocks = 11240
number of block colors = 4
maximum block size = 128
average thread colors = 4.00
shared memory required = 3.72 KB
average data reuse = 3.20
data transfer (used) = 87.13 MB
data transfer (total) = 143.06 MB

- factor 2-4 data reuse in indirect access, but up to 40% of cache lines not used on average
Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using initial parameter values:

- mini-partition size (PS): 256 elements
- blocksize (BS): 256 threads

<table>
<thead>
<tr>
<th>count</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.23</td>
<td>107.8</td>
<td></td>
<td>save_soln</td>
</tr>
<tr>
<td>2000</td>
<td>1.26</td>
<td>61.0</td>
<td>63.1</td>
<td>adt_calc</td>
</tr>
<tr>
<td>2000</td>
<td>5.10</td>
<td>32.5</td>
<td>53.4</td>
<td>res_calc</td>
</tr>
<tr>
<td>2000</td>
<td>0.11</td>
<td>4.8</td>
<td>18.4</td>
<td>bres_calc</td>
</tr>
<tr>
<td>2000</td>
<td>1.07</td>
<td>110.6</td>
<td></td>
<td>update</td>
</tr>
<tr>
<td>TOTAL</td>
<td>7.78</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Second B/W column includes whole cache line
Airfoil test code

Single precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

<table>
<thead>
<tr>
<th>count</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
<th>PS</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.22</td>
<td>101.8</td>
<td></td>
<td>save_soln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>1.09</td>
<td>74.1</td>
<td>75.4</td>
<td>adt_calc</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>4.95</td>
<td>36.9</td>
<td>60.6</td>
<td>res_calc</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>0.10</td>
<td>5.3</td>
<td>20.0</td>
<td>bres_calc</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>1.03</td>
<td>94.7</td>
<td></td>
<td>update</td>
<td></td>
<td>64</td>
</tr>
<tr>
<td>TOTAL</td>
<td>7.40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is a 5 % improvement relative to baseline calculation. Switching from AoS to SoA storage would increase res_calc data transfer by approximately 120%.
Airfoil test code

Double precision performance for 1000 iterations on an NVIDIA C2070 using auto-tuned values:

<table>
<thead>
<tr>
<th>count</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
<th>PS</th>
<th>BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>0.44</td>
<td>104.9</td>
<td></td>
<td>save_soln</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>2.62</td>
<td>52.9</td>
<td>53.8</td>
<td>adt_calc</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>10.35</td>
<td>30.5</td>
<td>50.8</td>
<td>res_calc</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>0.08</td>
<td>11.2</td>
<td>27.9</td>
<td>bres_calc</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>1.87</td>
<td>104.5</td>
<td></td>
<td>update</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TOTAL  | 15.36 |      |      |              |     |    |

This is a 7.5 % improvement relative to baseline calculation.
Switching from AoS to SoA storage would again increase res_calc data transfer by approximately 120%.
Airfoil test code

Single precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 16

<table>
<thead>
<tr>
<th>count</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.68</td>
<td>13.7</td>
<td></td>
<td>save_soln</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>11.15</td>
<td>7.3</td>
<td>7.5</td>
<td>adt_calc</td>
<td>128</td>
</tr>
<tr>
<td>2000</td>
<td>16.57</td>
<td>10.3</td>
<td>11.2</td>
<td>res_calc</td>
<td>1024</td>
</tr>
<tr>
<td>2000</td>
<td>0.16</td>
<td>3.2</td>
<td>11.9</td>
<td>bres_calc</td>
<td>64</td>
</tr>
<tr>
<td>2000</td>
<td>4.67</td>
<td>20.9</td>
<td></td>
<td>update</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>34.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.
Airfoil test code

Double precision performance on two Intel “Westmere” 6-core 2.67GHz X5650 CPUs using auto-tuned values:

Optimum number of OpenMP threads: 12

<table>
<thead>
<tr>
<th>count</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
<th>PS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2.51</td>
<td>18.3</td>
<td></td>
<td>save_soln</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>11.68</td>
<td>11.8</td>
<td>11.9</td>
<td>adt_calc</td>
<td>1024</td>
</tr>
<tr>
<td>2000</td>
<td>20.99</td>
<td>12.8</td>
<td>13.5</td>
<td>res_calc</td>
<td>1024</td>
</tr>
<tr>
<td>2000</td>
<td>0.17</td>
<td>5.0</td>
<td>12.4</td>
<td>bres_calc</td>
<td>512</td>
</tr>
<tr>
<td>2000</td>
<td>9.29</td>
<td>21.1</td>
<td></td>
<td>update</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>44.64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Minimal gain relative to baseline calculation with 12 threads and mini-partition sizes of 1024.
Conclusions

- have created a high-level framework for parallel execution of unstructured grid algorithms on GPUs and other many-core architectures
- looks encouraging for providing ease-of-use, high performance and longevity through new back-ends
- auto-tuning is useful for code optimisation, and a new flexible auto-tuning system has been developed
- C2070 GPU speedup versus two 6-core Westmere CPUs is roughly $5 \times$ in single precision, $3 \times$ in double precision
- currently working on MPI layer in OP2 for computing on GPU clusters
- key challenge then is to build user community