Finite Element Methods in OP2 for heterogeneous architectures

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

Oxford e-Research Centre, Imperial College London, PPKE ITK Hungary

ECCOMAS 2012: MS404-2
OP2 – abstraction of unstructured grids

• Collection of sets
  – Edges, vertices, cells
• Connectivity
  – Explicit, unlike structured grids
  – Mapping tables between sets
• Data on sets
  – Pressure, velocity, etc.
• Operations over sets accessing data through mappings
Software Challenge

• Developers want the benefits of the latest hardware but are worried about development costs
  – CUDA, OpenCL, OpenMP, AVX combined with MPI
• Different/future hardware requires different code optimizations

Solution?
• High-level abstraction to separate user specification from parallel implementation
• Thereby achieving code longevity and near-optimal performance by swapping backends
• Open source project
  – http://www.oerc.ox.ac.uk/research/op2
• Based on OPlus (Oxford Parallel Library for Unstructured Solvers) developed over 15 years ago for industrial CFD code on distributed-memory clusters
• Supports application codes written in C++ or FORTRAN
• Keeps OPlus abstraction, but slightly modifies API
• Looks like a conventional library, but uses code transformation to generate CUDA for NVIDIA GPUs and OpenMP/AVX for CPUs/MIC
• FEniCS integration
OP2 Abstraction

• Built around the notion of operations over sets
  – Loop over the members of a set, accessing data with at most one level of indirection (through maps) in a way defined by the user (read/write/increment)

• Key parallelization strategy: set elements can be executed in any order
  – Jacobi iteration is okay, Gauss-Seidel is not.

• Sets and maps do not change
  – No mesh adaptation
OP2 API

• Opaque types for sets, maps and data
• Example of a parallel loop (spMV)

```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));
```

Name
Loop over set
Read nonzeros (edges)
Read vertex value on one end of the edge
Increment value on the other
Build process

• User implements serial code in C++/Fortran using the OP2 API
  – Easy development and debugging

• Code transformation to different backends
  – CUDA (and OpenCL in the future)
  – OpenMP (and AVX in the future)
  – MPI combined with any of the above
Levels of parallelism

• MPI distributed-memory parallelism (1-100)
  – Partitioning, halo exchanges between partitions, latency hiding etc.

• Block parallelism (100-2000)
  – Data broken into mini-partitions and colored for parallel execution

• Fine-grained parallelism (64-256)
  – When executed on the GPU, threads process different set elements and increment indirectly referenced data color-by-color
Techniques used

• Coloring to handle race conditions
  – Mini-partitions: coarse grain parallelism
  – Set elements: fine grain parallelism

• Array of structures data layout
  – Better cache hit rates for indirect addressing

• Auto-tuning

• MPI latency-hiding

• Checkpointing
Finite Elements in OP2

• Uses the notion of sets and mappings between sets

• Create sparse linear system
  – Node-centered approach not supported as it would require fixed degree vertices
  – Assembly cell-by-cell, indirect increment of coefficients

• Solve sparse linear system
  – Send off to external solver
  – Use the API
Assembly – data locality

Preload input data in scratch-pad memory

for each pair of d.o.f.
  for each quadrature point
    update nonzero

vs.

for each quadrature point
  for each pair of d.o.f.
    update nonzero

All the nonzeros of all the cells processed in parallel may not fit in cache of the GPU
Assembly

• Global Matrix Assembly into CSR/ELLPACK
  – Expensive mapping table (precompute layout)
  – Has to be handed off to separate solver

• Matrix-free method (LMA)
  – Local stiffness matrix for every cell
  – No data races
  – Can be solved with OP2
Performance of Poisson assembly

12 core Westmere X5650 vs Tesla C2070

Time of assembly (seconds)

OpenMP threads

CSR
ELLPACK
LMA

C2070
Linear solve with OP2

• With the OP2 constraints we cannot do CSR spMV, since the number of non-zeros per row is different
• ELLPACK is aligned, but rows are still too long for one set element
• LMA is per cell, thus easy to implement
• Conjugate-gradient iteration
  – One spMV, 5 BLAS operations
  – “Assembly” happens during spMV
Performance of LMA solve

Time of 1 CG iteration (seconds)

Westmere 2*6-core X5650, 2.67 GHz
Aero test code

• 2D non-linear steady potential flow simulation of air moving around an airfoil
• Quadrilateral grid, 720K/12M/26M cells
• Newton iteration
  – Assembly using FEM
  – Symmetric sparse linear system solved with conjugate gradient iteration
• Matrix-free assembly/solve
  – Easy to exploit symmetry
• Double precision
Newton iteration

\[
\text{op\_par\_loop( assembly )} \quad 9% \\
\text{op\_par\_loop( apply\_dirichlet )} \\
\text{op\_par\_loop( init\_CG) }
\]
while (stopping condition) //~200 iterations

\[
\text{op\_par\_loop( spMV )} \quad 60% \\
\text{op\_par\_loop( apply\_dirichlet )} \\
\text{op\_par\_loop( dotPV )} \\
\text{op\_par\_loop( updateUR )} \\
\text{op\_par\_loop( dotRR )} \\
\text{op\_par\_loop( updateP )} \quad 30%
\]
end while

\[
\text{op\_par\_loop( update\_solution )}
\]
end Newton iteration
### Aero single node spMV

<table>
<thead>
<tr>
<th>System</th>
<th>MPI x OpenMP</th>
<th>Time (sec)</th>
<th>GB/sec (useful)</th>
<th>GB/sec (total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Westmere</td>
<td>1*24</td>
<td>31.09</td>
<td>19.28</td>
<td>19.41</td>
</tr>
<tr>
<td></td>
<td>12*2</td>
<td>20.72</td>
<td>27.5</td>
<td>28.1</td>
</tr>
<tr>
<td>Interlagos</td>
<td>1*32</td>
<td>60.69</td>
<td>9.87</td>
<td>9.94</td>
</tr>
<tr>
<td></td>
<td>4*8</td>
<td>11.28</td>
<td>53.32</td>
<td>54.48</td>
</tr>
<tr>
<td>C2090</td>
<td>CUDA</td>
<td>5.79</td>
<td>103.52</td>
<td>105.37</td>
</tr>
</tbody>
</table>

Westmere: 2*6-core X5650, 2.67 GHz
Interlagos: 2*16-core Opteron 6276, 2.3 GHz
C2090: 512 CUDA cores, 1.15GHz, ECC on
Aero scaling – 720k

![Graph showing execution time vs number of nodes for different systems and configurations. The systems include CX1 (12 MPI), CX1 (12 MPI x 2 OMP), HECToR (32 MPI), HECToR (4 MPI x 8 OMP), and EMERALD (MPI x 3 GPU). The execution time is measured in seconds and the number of nodes ranges from 1 to 128.]
Aero scaling – 12M

Execution time (Seconds)

Number of nodes

CX1 (12 MPI)
CX1 (12 MPI x 2 OMP)
HECToR (32 MPI)
HECToR (4 MPI x 8 OMP)
EMERALD (MPI x 3 GPU)
Aero scaling – 26M

Execution time (Seconds)

Number of nodes

CX1 (12 MPI)
CX1 (12 MPI x 2 OMP)
HECToR (32 MPI)
HECToR (4 MPI x 8 OMP)
EMERALD (MPI x 3 GPU)
Conclusions & Future work

• A high-level framework for parallel execution of unstructured-grid algorithms on heterogeneous systems
• Providing ease-of-use, high performance and code longevity through new back-ends
• 1 C2090 about 3 times faster than 12-core Xeon X5650 in double precision

• Abstracting the loop over pairs of degrees of freedom
  – More control over granularity, more parallelism
• Better global matrix assembly support
GMA/LMA spMV memory traffic

- “Uniform” quadrilateral grid

<table>
<thead>
<tr>
<th>degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELL/LMA sp</td>
<td>1.0</td>
<td>1.81</td>
<td>2.25</td>
<td>2.49</td>
</tr>
<tr>
<td>CSR/LMA sp</td>
<td>1.04</td>
<td>1.85</td>
<td>2.28</td>
<td>2.51</td>
</tr>
<tr>
<td>ELL/LMA dp</td>
<td>0.9</td>
<td>1.58</td>
<td>1.93</td>
<td>2.12</td>
</tr>
<tr>
<td>CSR/LMA dp</td>
<td>0.92</td>
<td>1.6</td>
<td>1.95</td>
<td>2.13</td>
</tr>
</tbody>
</table>

2D

<table>
<thead>
<tr>
<th>degree</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELL/LMA sp</td>
<td>0.93</td>
<td>1.9</td>
<td>2.36</td>
<td>2.59</td>
</tr>
<tr>
<td>CSR/LMA sp</td>
<td>0.94</td>
<td>1.91</td>
<td>2.37</td>
<td>2.6</td>
</tr>
<tr>
<td>ELL/LMA dp</td>
<td>0.81</td>
<td>1.61</td>
<td>1.98</td>
<td>2.17</td>
</tr>
<tr>
<td>CSR/LMA dp</td>
<td>0.82</td>
<td>1.62</td>
<td>1.99</td>
<td>2.17</td>
</tr>
</tbody>
</table>

3D
res_calc/spmv stats

new execution plan #0 for kernel res_calc
number of blocks = 5625
number of block colors = 4
maximum block size = 128
average thread colors = 2.00
shared memory required = 8.06 KB, 8.12 KB, 8.12 KB, 8.12 KB
average data reuse = 1.98
data transfer (used) = 264.26 MB
data transfer (total) = 269.27 MB
SoA/AoS transfer ratio = 1.01

new execution plan #2 for kernel spMV
number of blocks = 5625
number of block colors = 4
maximum block size = 128
average thread colors = 2.00
shared memory required = 4.03 KB, 4.06 KB, 4.06 KB, 4.06 KB
average data reuse = 1.98
data transfer (used) = 143.20 MB
data transfer (total) = 147.04 MB
SoA/AoS transfer ratio = 1.00