Tsunami simulation using the OP2 parallel framework

Mike Giles, Endre Laszlo, Gihan Mudalige, István Reguly (Oxford) Serge Guillas (UCL), Carlo Bertolli (IBM), Paul Kelly (Imperial College)

Oxford e-Research Centre

25th Biennial Conference on Numerical Analysis

June 25th, 2013
VOLNA

- VOLNA is a Tsunami simulation code developed by Denys Dutykh, Raphaël Poncet and Frédéric Dias, and used by Serge Guillas at UCL.
- Modelling uses 2D shallow water equations, discretised on triangular meshes.
- 3 state variables per cell:
  - Water height $h$
  - Velocity components $u, v$
- Bathymetry (height of sea-bed) is also stored, and modelling allows for wetting as waves run up onto dry ground.

VOLNA = wave in Russian.
VOLNA

Tsunami simulation at Catalina Island in California
Grid for Alaskan Aleutian islands
VOLNA

Original code was written in C++

- strongly based on object-oriented programming (OOP)
- used STL and Boost C++ libraries
- strong emphasis on clarity of programming, and flexibility to experiment with new algorithms
- no parallel capability
- in some places OOP approach led additional data movement, with large temporary arrays being constructed
- our goal was to develop a new parallel version supporting
  - OpenMP parallelism on multicore x86 CPUs
  - CUDA parallelism on NVIDIA GPUs
  - MPI distributed memory parallelism for clusters of CPUs or GPUs
  - parallel file I/O
- total development effort – about 3 man-months using OP2
Problem:

- application developers want the benefits of the latest hardware but are very worried about the development effort required
- want to exploit multicore CPUs (using OpenMP) and many-core GPUs (using CUDA), and also run on clusters
- however, hardware is likely to change rapidly in 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 a collaboration between Oxford and Imperial College
based on OPlus (Oxford Parallel Library for Unstructured Solvers) developed over 15 years ago for a Rolls-Royce 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 for CPUs
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

```c
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.
Example of one parallel loop in VOLNA:

```c
op_par_loop(SpaceDiscretization, "SpaceDisc", edges,
    op_arg_dat(data_out, 0, edgesToCells, 4, "float", OP_INC),
    op_arg_dat(data_out, 1, edgesToCells, 4, "float", OP_INC),
    op_arg_dat(edgeFluxes, -1, OP_ID, 3, "float", OP_READ),
    op_arg_dat(bathySource, -1, OP_ID, 2, "float", OP_READ),
    op_arg_dat(edgeNormals, -1, OP_ID, 2, "float", OP_READ),
    op_arg_dat(isBoundary, -1, OP_ID, 1, "int", OP_READ),
    op_arg_dat(cellVolumes, -2, edgesToCells, 1, "float", OP_READ)
);
```
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 for single GPU

- OpenMP 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`
- `jac_op.cpp`
- `jac_kernels.cu`
- `res_kernel.cu`
- `update_kernel.cu`
- `op_lib.cu`

- python preprocessor

- `make / nvcc / g++`
Implementation Approach

Standard MPI distributed-memory parallelism:
- Parmetis or PT-SCOTCH for grid partitioning and local renumbering
- separate MPI process for each GPU, with each partition fitting within the GPUs global memory

GPU parallelism:
- partition sub-divided into blocks, each handled separately
- blocks sized to fit in shared memory for data reuse
- block colouring used to avoid data conflicts between blocks, and thread colouring to avoid conflicts within a block

CPU parallelism:
- similar block construction, with each handled by one OpenMP thread
- block colouring is again used to prevent data conflicts between blocks
Porting VOLNA to OP2

- a complete re-structuring of the code – almost no option for incremental transformation
- key numerical bits, such as numerical flux functions, kept as is
- overall time-marching control code largely the same as before
- still as readable as before (?)
- input/output data files converted to HDF5 for parallel file I/O
- VTK viewer developed for post-processing
- original input control file retained to minimise change for users
Porting VOLNA to OP2

One issue which came up was the need for better local renumbering

- on GPUs, original numbering led to poorly shaped blocks with lots of neighbours
  - up to 72 block colours required
  - poor cache efficiency (a lot of each cache line loaded not used)
- it also led to poor cache efficiency on CPUs
- improved renumbering reduced block colours to 6 on GPU, and reduced data transfer from main memory to the GPU/CPU

Another issue on CPUs was the need for enforcing thread affinity, to tie threads to the same core to maximise data reuse in the cache.
VOLNA performance

Test machine: \(2 \times 6\)-core Xeon (Westmere) + NVIDIA C2070 GPU

2 CPUs versus 1 GPU (different number of timesteps in each case)

<table>
<thead>
<tr>
<th>Examples</th>
<th>Cells / edges</th>
<th>OpenMP</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Landslide (synthetic)</td>
<td>500k / 751k</td>
<td>18.4s</td>
<td>7.5s</td>
</tr>
<tr>
<td>Catalina (1993)</td>
<td>98k / 147k</td>
<td>1.2s</td>
<td>0.74s</td>
</tr>
<tr>
<td>Mentawi (2012)</td>
<td>140k / 211k</td>
<td>2.0s</td>
<td>1.05s</td>
</tr>
<tr>
<td>Vancouver (hypothetical)</td>
<td>2.4M / 3.6M</td>
<td>5.15s</td>
<td>2.1s</td>
</tr>
</tbody>
</table>

Hence, on big testcases, 1 GPU is equivalent to roughly 5 CPUs
VOLNA performance

VOLNA is memory bandwidth limited on both CPUs and GPUs.

<table>
<thead>
<tr>
<th>2xCPU</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.37</td>
<td>30</td>
<td>0.37</td>
<td>30</td>
<td>EvolveRK</td>
</tr>
<tr>
<td>1.41</td>
<td>24</td>
<td>1.41</td>
<td>24</td>
<td>ComputeFluxes</td>
</tr>
<tr>
<td>0.74</td>
<td>29</td>
<td>0.74</td>
<td>30</td>
<td>NumericalFluxes</td>
</tr>
<tr>
<td>1.46</td>
<td>25</td>
<td>1.46</td>
<td>25</td>
<td>SpaceDiscretization</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1xGPU</th>
<th>time</th>
<th>GB/s</th>
<th>GB/s</th>
<th>kernel name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.18</td>
<td>105</td>
<td>0.18</td>
<td>105</td>
<td>EvolveRK</td>
</tr>
<tr>
<td>0.59</td>
<td>58</td>
<td>0.59</td>
<td>58</td>
<td>ComputeFluxes</td>
</tr>
<tr>
<td>0.28</td>
<td>79</td>
<td>0.28</td>
<td>79</td>
<td>NumericalFluxes</td>
</tr>
<tr>
<td>0.69</td>
<td>52</td>
<td>0.69</td>
<td>52</td>
<td>SpaceDiscretization</td>
</tr>
</tbody>
</table>

Second B/W column includes whole cache line
VOLNA performance

Test machine: Emerald GPU cluster with compute nodes with 6-core Intel Xeons (Westmere) + NVIDIA C2090 GPUs (Fermi)

![Graph showing performance comparison between CPU and GPU across different procs]

- X-axis: procs
- Y-axis: time
- Blue line: CPU
- Red line: GPU
VOLNA performance

Strong scaling is quite good on Vancouver testcase (2.4M cells) — on other OP2 applications, weak scaling is usually better.
Conclusions

- excellent scalable parallel performance has been achieved for an important tsunami simulation code

- development effort was low using OP2 framework, and code readability was maintained

- VOLNA-OP2 will now be used for new tsunami research at UCL in collaboration with Indian Institute of Science in Bangalore

- OP2 challenge is to get someone to port a major code on their own – can only really claim a tool is easy-to-use if someone else uses it!

- new EPSRC project with Bristol, Southampton and STFC will address needs of multi-block structured grid codes with a focus on two particular CFD codes
References


OP2 project homepage: http://www.oerc.ox.ac.uk/research/op2