OP2 Airfoil Example

Mike Giles, Gihan Mudalige, István Reguly

December 2013

Contents

1 Introduction

Airfoil, is an industrial representative CFD application benchmark, written using OP2's C/C++ API. In this document we detail its development using OP2 as a guide to application developers wishing to write applications using the OP2 API and framework.

Full details of OP2 can be found at: http://www.oerc.ox.ac.uk/research/op2

Airfoil is a non-linear 2D inviscid airfoil code that uses an unstructured grid. It is a finite volume application that solves the 2D Euler equations using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach - for example the rate at which the mass changes within a control volume is equal to the net flux of mass into the control volume across the four faces around the cell. This is representative of the 3D viscous flow calculations OP2 supports for production-grade CFD applications (such as the Hydra [1, 2] CFD code at Rolls Royce plc.). Airfoil consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc` and `update`. Out of these, `save_soln` and `update` are direct loops while the other three are indirect loops. The standard mesh size solved with Airfoil consists of 1.5M edges. In such a mesh the most compute intensive loop, `res_calc`, is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge. Extensive performance analysis of Airfoil and optimisations have been detailed in our published work [3, 4, 5, 6]. What follows is a step-by-step treatment of the stages involved in developing Airfoil. The application and generated code can be found under `OP2-Common/apps/c/airfoil`.

2 Airfoil - The Development CPU Version

For the application developer wishing to utilise the OP2 framework for developing unstructured mesh codes, the first step is to develop the application assuming that the target execution system is a traditional single threaded CPU. This simplifies the programming complexity of the code by allowing the developer to concentrate on the application domain and science involved and not the intricacies of parallel programming. During development the code can be run without any OP2 code generation, on a single CPU thread by simply including the `op_seq.h` header file.

The Airfoil application level code (i.e. the code written by the domain scientist) consists of only a main function (in `airfoil.cpp`) where it performs a time marching loop that executes 1000 times, each iteration calling the above mentioned five loops. Each loop iterates over a specified `op_set` and the operations to be performed per iteration are detailed in a header file for each loop: `save_soln.h, adt_calc.h, res_calc.h, bres_calc.h` and `update.h`. The following code illustrates the declaration of the `res_calc` loop, which iterates over the edges of the mesh.

```
//  calculate flux residual
op_par_loop(res_calc,"res_calc",edges,
            op_arg_dat(p_x,    0,pedge, 2,"double",OP_READ),
            op_arg_dat(p_x,    1,pedge, 2,"double",OP_READ),
            op_arg_dat(p_q,    0,pecell,4,"double",OP_READ),
            op_arg_dat(p_q,    1,pecell,4,"double",OP_READ),
            op_arg_dat(p_adt,  0,pecell,1,"double",OP_READ),
            op_arg_dat(p_adt,  1,pecell,1,"double",OP_READ),
            op_arg_dat(p_res,  0,pecell,4,"double",OP_INC ),
            op_arg_dat(p_res,  1,pecell,4,"double",OP_INC ));
```

Figure 1: res_calc loop

```
// OP2 header file
#include "op_seq.h"
// global constants
double gam, gm1, cfl, eps, mach, alpha, qinf[4];
// user kernel routines for parallel loops
#include "save_soln.h"
#include "adt_calc.h"
#include "res_calc.h"
#include "bres_calc.h"
#include "update.h"
```

Figure 2: Header and global constants

The first argument specifies the name of the function (implemented in `res_calc.h`) that contains the operations to be performed per iteration in the loop. The second argument notes the name of the loop, while the third specifies the `op_set` over which the loop will iterate. The remaining arguments specify the access descriptors of the data used in the loop. More details of the `op_arg_dat` API statements are given in the user guide.

For Airfoil, `airfoil.cpp` include the OP2 header files and the elemental kernel header files as follows. Additionally global constants must be declared before the `main()` function.

The main function of an OP2 applications need to begin with the initialisation statement `op_init()` before any other OP2 API statements can be called. The Airfoil application then reads in the input mesh file. As detailed in the user documentation, OP2 allows the application developer to carry out their own I/O or utilise hdf5 based I/O using OP2's hdf5 API statements. We first detail the development of Airfoil assuming that file I/O is implemented by the application developer. Later in Section 5 we will detail the Airfoil application written utilising OP2's hdf5 file I/O capabilities.

Assume that the application reads an ASCI file using standard C++ file I/O statements to read in the input mesh and allocate memory to hold the data. The data is then passed

to the appropriate OP2 declaration statements to declare op_sets, op_maps and op_dats as follows:

```
op_set nodes  = op_decl_set(nnode,  "nodes");
op_set edges  = op_decl_set(nedge,  "edges");
op_set bedges = op_decl_set(nbedge, "bedges");
op_set cells  = op_decl_set(ncell,  "cells");

op_map pedge   = op_decl_map(edges, nodes,2,edge,  "pedge");
op_map pecell  = op_decl_map(edges, cells,2,ecell, "pecell");
op_map pbedge  = op_decl_map(bedges,nodes,2,bedge, "pbedge");
op_map pbecell = op_decl_map(bedges,cells,1,becell,"pbecell");
op_map pcell   = op_decl_map(cells, nodes,4,cell,  "pcell");

op_dat p_bound = op_decl_dat(bedges,1,"int"   , bound ,"p_bound");
op_dat p_x     = op_decl_dat(nodes ,2,"double", x     ,"p_x");
op_dat p_q     = op_decl_dat(cells ,4,"double", q     ,"p_q");
op_dat p_qold  = op_decl_dat(cells ,4,"double", qold  ,"p_qold");
op_dat p_adt   = op_decl_dat(cells ,1,"double", adt   ,"p_adt");
op_dat p_res   = op_decl_dat(cells ,4,"double", res   ,"p_res");

op_decl_const(1,"double",&gam  );
op_decl_const(1,"double",&gm1  );
op_decl_const(1,"double",&cfl  );
op_decl_const(1,"double",&eps  );
op_decl_const(1,"double",&mach );
op_decl_const(1,"double",&alpha);
op_decl_const(4,"double",qinf  );
```

Figure 3: OP2 set, map and dat declarations

Four sets are declared (nodes, edges, bedges and cells) and five mappings between sets are declared to establish connectivity between sets. The six data arrays are declared on the sets bedges, nodes and cells. Any constants used by the program are also declared at this point using op_decl_const(). The five parallel loops that make up the Airfoil application are detailed next within a time-marching loop.

Finally, statistics about the performance of the application can be printed to stdout using op_timing_output() and OP2 is terminated with op_exit(), deallocating internal OP2 allocated memory. If the application developer allocated memory, for example to read in the mesh, then he/she will need to manually deallocate memory at this point.

During development, by simply including the op_seq.h header file application can be compiled by a conventional compiler (gcc, icc etc.) to produce an executable that can be run on a single threaded CPU. No code generation is required at this stage, where the application

```
//main time-marching loop
for(int iter=1; iter<=niter; iter++) {
    // save old flow solution
    op_par_loop(save_soln, ... );

    // predictor/corrector update loop
    for(int k=0; k<2; k++) {

      // calculate area/timstep
      op_par_loop(adt_calc, ... );

      // calculate flux residual
      op_par_loop(res_calc, ... );

      op_par_loop(bres_calc, ... );

      // update flow field
      op_par_loop(update, ... );
    }
    ...
}
```

Figure 4: Time marching loop

developer is only using the single threaded executable for debugging and development. This enable the developer to build the application and test its accuracy without considering any parallelisation issues. The compilation of the application for single-threaded CPUs is achieved by linking with the OP2 sequential library `libop2_seq.a`, for example as follows:

```
OP2_INC         = -I$(OP2_INSTALL_PATH)/c/include
OP2_LIB         = -L$(OP2_INSTALL_PATH)/c/lib
CPP             = icpc
CPPFLAGS        = -O3 -xSSE4.2
airfoil_seq: airfoil.cpp save_soln.h adt_calc.h res_calc.h \
             bres_calc.h update.h
             $(CPP) $(CPPFLAGS) airfoil.cpp \
             $(OP2_INC) $(OP2_LIB) -lop2_seq -o airfoil_seq
```

Figure 5: Sequential developer version build

Once the application is debugged and tested on a single CPU, OP2's code generation capabilities can be used to generate executables for different parallel architectures. We will use OP2's Python code parser for code generation throughout this document. Specific details of the build process for each target back-end is detailed in the next sections.

3 Generating Single Node OpenMP and CUDA Executables

This section will detail code generation for a single CPU node (SMP or CMP node) or a single GPU, targeting OpenMP and NVIDIA CUDA respectively. First, the code generator needs to be invoked, for example for parsing `airfoil.cpp`:

$ > ./op2.py airfoil.cpp

The code generator will produce a modified main program and back-end specific code. In this case `airfoil.cpp` will be transformed to `airfoil_op.cpp` and a kernel file will be produced corresponding to each `op_par_loop` in the main program (`*_kernel.cpp` for OpenMP and `*_kernel.cu` for CUDA). For running on SMP CPU nodes, OpenMP is utilised. The executable can be built by compiling the code with a conventional C++ compiler and linking with the openmp back-end library, `libop2_openmp.a`, for example as follows:

```
OP2_INC         = -I$(OP2_INSTALL_PATH)/c/include
OP2_LIB         = -L$(OP2_INSTALL_PATH)/c/lib
CPP             = icpc
CPPFLAGS        = -O3 -xSSE4.2
OMPFLAGS        = -openmp -openmp-report2
airfoil_openmp:       airfoil_op.cpp airfoil_kernels.cpp \
                save_soln_kernel.cpp save_soln.h \
                adt_calc_kernel.cpp adt_calc.h \
                res_calc_kernel.cpp res_calc.h \
                bres_calc_kernel.cpp bres_calc.h \
                update_kernel.cpp update.h
                $(CPP) $(CPPFLAGS) $(OMPFLAGS) $(OP2_INC) $(OP2_LIB) \
                airfoil_op.cpp airfoil_kernels.cpp \
                -lm -lop2_openmp -o airfoil_openmp
```

Figure 6: OpenMP version build

The `airfoil_kernels.cpp` includes all the `*_kernel.cpp` files. This is why it is the only file appearing in the compile line. The `airfoil_openmp` can be run on multiple OpenMP threads by setting the `OMP_NUM_THREADS` environmental variable.

For running on a single GPU, NVIDIA CUDA is utilised. The executable can be built by compiling with nvcc and a conventional C++ compiler and linking with the CUDA back-end library, `libop2_cuda.a`, for example as follows:

```
OP2_INC       = -I$(OP2_INSTALL_PATH)/c/include
OP2_LIB       = -L$(OP2_INSTALL_PATH)/c/lib
CPP           = icpc
CPPFLAGS       = -O3 -xSSE4.2
CUDA_INC      = -I$(CUDA_INSTALL_PATH)/include
CUDA_LIB      = -L$(CUDA_INSTALL_PATH)/lib64
NVCCFLAGS     = -O3 -arch=sm_20 -Xptxas=-v -Dlcm=ca -use_fast_math
airfoil_cuda:       airfoil_op.cpp airfoil_kernels_cu.o
              $(CPP) $(CPPFLAGS) $(CUDA_INC) $(OP2_INC) \
              $(OP2_LIB) $(CUDA_LIB) \
              airfoil_op.cpp airfoil_kernels_cu.o -lcudart \
             -lop2_cuda -o airfoil_cuda

airfoil_kernels_cu.o:       airfoil_kernels.cu     \
              save_soln_kernel.cu save_soln.h \
              adt_calc_kernel.cu  adt_calc.h  \
              res_calc_kernel.cu  res_calc.h  \
              bres_calc_kernel.cu bres_calc.h \
              update_kernel.cu    update.h
              nvcc $(NVCCFLAGS) $(OP2_INC) \
              -c -o airfoil_kernels_cu.o \
              airfoil_kernels.cu
```

Figure 7: CUDA version build

Similar to the OpenMP compilation, the `airfoil_kernels.cu` includes all the `*_kernel.cu` files. When `airfoil_cuda` is executed, it will select any available GPU on the system. The GPU to be selected can be set by using the `CUDA_VISIBLE_DEVICES` environment variable. The steps are illustrated in Figure 8.

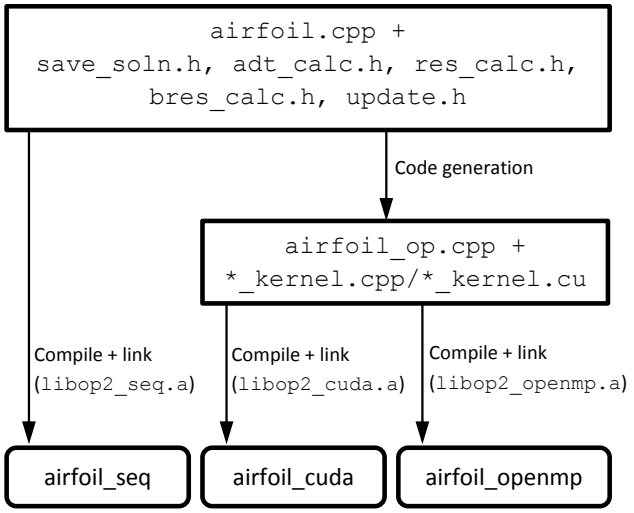

Figure 8: Single-node Code generation and build for Airfoil (with user I/O)

4 Building Airfoil for Distributed Memory (MPI) Execution

If the application developer decides to be responsible for the application's I/O, i.e. for reading in the unstructured mesh, then for distributed memory execution of the application, parallelising the I/O process cannot be simply automated. As such OP2's code generation tools does not support generating an MPI based application, by simply parsing `airfoil.cpp`. What is required is the development of `airfoil_mpi.cpp` that explicitly codes the parallel I/O. The only difference between `airfoil.cpp` and `airfoi_mpi.cpp` is that the latter hands partitions of the `op_set`s, `op_map`s and `op_dat`s that resides on each MPI process to OP2 via `op_decl_*` statements (see `OP2-Common/airfoil/dp/airfoi_mpi.cpp`). OP2 supports the development of such an MPI application, without the need for any code generation. Similar to the development process for a single threaded CPU, all that is required is to include the `op_seq.h` header file. The MPI application can be built by compiling with mpiCC and linking with the MPI back-end library, `libop2_mpi.a`, for example as follows:

The unstructured mesh, will be repartitioned by OP2, using parallel graph/mesh partitioning libraries (ParMetis or PTScotch) as detailed in the user documentation. Thus linking with the appropriate mesh partitioning library will also be needed as detailed above. The resulting executable, `airfoil_mpi`, can be executed on a cluster of single threaded CPUs, with the use of the usual mpirun command.

```
MPICPP          = mpiCC
MPIFLAGS        = -O3 -xSSE4.2

PARMETIS_INC    = -I$(PARMETIS_INSTALL_PATH) -DHAVE_PARMETIS
PARMETIS_LIB    = -L$(PARMETIS_INSTALL_PATH) -lparmetis \
                  -L$(PARMETIS_INSTALL_PATH) -lmetis

PTSCOTCH_INC    = -I$(PTSCOTCH_INSTALL_PATH)/include -DHAVE_PTSCOTCH
PTSCOTCH_LIB    = -L$(PTSCOTCH_INSTALL_PATH)/lib/ -lptscotch \
                  -L$(PTSCOTCH_INSTALL_PATH)/lib/ -lptscotcherr

airfoil_mpi: airfoil_mpi.cpp \
        save_soln.h adt_calc.h res_calc.h bres_calc.h Makefile
        $(MPICPP) $(MPIFLAGS) $(OP2_INC) \
        $(PARMETIS_INC) $(PTSCOTCH_INC) \
        $(OP2_LIB) airfoil_mpi.cpp -lop2_mpi \
        $(PARMETIS_LIB) $(PTSCOTCH_LIB) -o airfoil_mpi
```

Figure 9: MPI version build

Once developed, `airfoi_mpi.cpp` can be used to generate the code required to build the application for execution on distributed memory heterogeneous systems. Currently supported systems, are a cluster of multi-threaded CPUs (Using MPI and OpenMP) and a cluster of GPUs (using MPI and CUDA). The code generator needs to be invoked on the mpi program to generate MPI+OpenMP and MPI+CUDA versions of the application. For example, to generate distributed memory code from `airfoil_mpi.cpp`:

$ > ./op2.py airfoil_mpi.cpp

Similar to the single-node parallel version, the code generator will produce a modified main program and back-end specific code. The build steps are illustrated in Figure 12. In this case `airfoil_mpi.cpp` will be transformed to `airfoil_mpi_op.cpp` and a kernel file will be produced corresponding to each `op_par_loop` in the main program (`*_kernel.cpp` for OpenMP and `*_kernel.cu` for CUDA). By design these kernel files will be identical to the kernel files created for the single-node parallel back-ends. The executable for a cluster of multi-threaded CPUs can be built by compiling the code using a conventional C++ compiler and linking with the back-end library, `libop2_mpi.a` (see Figure 10). Note that the linking library is the same op2 library used when building the pure MPI version. This is simply due to the fact that there are no special functionality needed in the backend library to enable OpenMP parallelisam on top of MPI.

`airfoil_mpi_openmp` needs to be executed using mpirun and will utilise OMP_NUM_THREADS per MPI process on the multi-threaded CPU cluster during execution.

```
airfoil_mpi_openmp: airfoil_mpi_op.cpp airfoil_kernels.cpp \
                    save_soln_kernel.cpp  save_soln.h \
                    adt_calc_kernel.cpp   adt_calc.h  \
                    res_calc_kernel.cpp   res_calc.h  \
                    bres_calc_kernel.cpp  bres_calc.h \
                    update_kernel.cpp     update.h    \
                    Makefile
                    $(MPICPP) $(MPIFLAGS) $(OMPFLAGS) \
                    airfoil_mpi_op.cpp \
                    airfoil_kernels.cpp \
                    $(OP2_INC) $(PARMETIS_INC) $(PTSCOTCH_INC) \
                    $(OP2_LIB) -lop2_mpi \
                    $(PARMETIS_LIB) $(PTSCOTCH_LIB) -o airfoil_mpi_openmp
```

Figure 10: MPI+OpenMP version build

The executable for a cluster of GPUs can be built by compiling the code using a conventional C++ compiler, CUDA compiler nvcc and linking with the back-end library, `libop2_mpi_cuda.a`, for example as follows:

```
airfoil_mpi_cuda: airfoil_mpi_op.cpp airfoil_kernels_mpi_cu.o Makefile
                  $(MPICPP) $(MPIFLAGS) airfoil_mpi_op.cpp \
                  airfoil_kernels_mpi_cu.o \
                  $(OP2_INC) $(PARMETIS_INC) $(PTSCOTCH_INC) \
                  $(OP2_LIB) -lop2_mpi_cuda $(PARMETIS_LIB) \
                  $(PTSCOTCH_LIB) \
                  $(CUDA_LIB) -lcudart -o airfoil_mpi_cuda

airfoil_kernels_mpi_cu.o:       airfoil_kernels.cu      \
                  save_soln_kernel.cu  save_soln.h \
                  adt_calc_kernel.cu   adt_calc.h  \
                  res_calc_kernel.cu   res_calc.h  \
                  bres_calc_kernel.cu  bres_calc.h \
                  update_kernel.cu     update.h    \
                  Makefile
                  nvcc  $(INC) $(NVCCFLAGS) $(OP2_INC) \
                  -I $(MPI_INSTALL_PATH)/include \
                  -c -o airfoil_kernels_mpi_cu.o airfoil_kernels.cu
```

Figure 11: MPI+CUDA version build

`airfoil_mpi_cuda` needs to be executed using mpirun and will utilise one GPU per MPI process on the GPU cluster during execution.

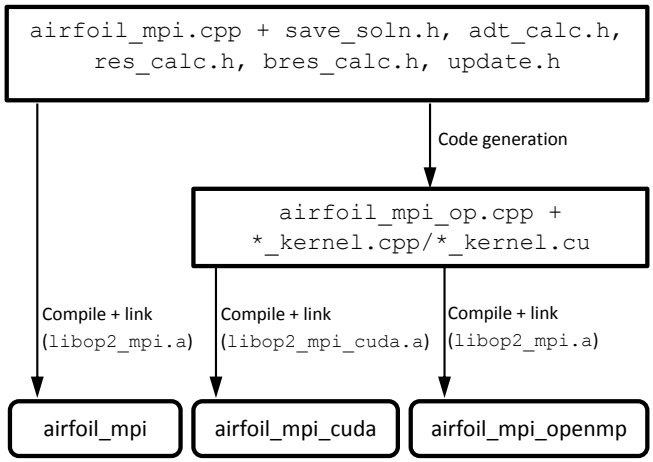

Figure 12: Distributed memory code generation and build for Airfoil (with user I/O)

5 Airfoil with HDF5 I/O

If OP2's file I/O is utilised when developing the application then all code required for **all** the target back-ends will be generated by parsing the `airfoil.cpp` file. This includes the distributed memory (MPI) back-ends. The code generation and application build process is summarised in Figure 13. The code generator produces a common modified main program in `airfoil_op.cpp` and kernel files, which is then linked with the appropriate OP2 back-ends libraries to give the desired target executable.

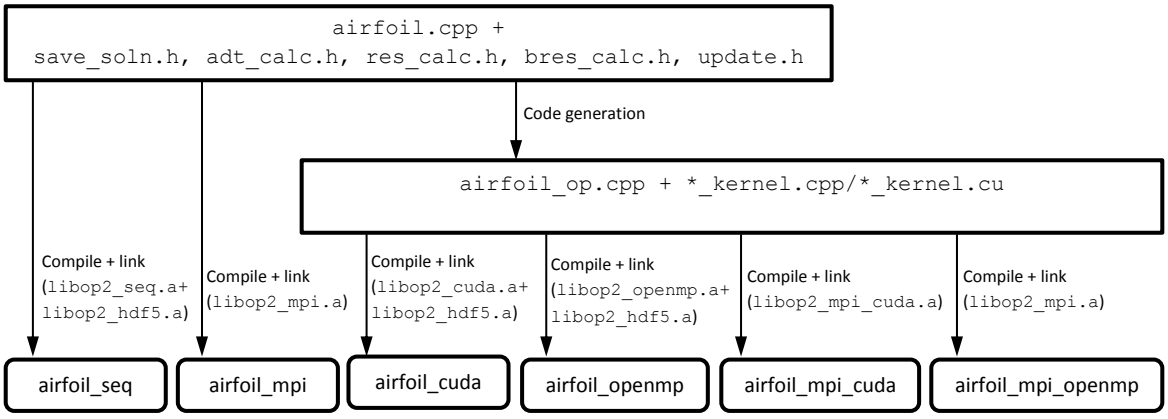

Figure 13: Code generation and build for Airfoil (with OP2 HDF5 file I/O)

The library `libop2_hdf5.a` needs to be linked when building single node executables, for example:

```
HDF5_INC = -I$(HDF5_INSTALL_PATH)/include
HDF5_LIB = -L$(HDF5_INSTALL_PATH)/lib -lhdf5 -lz

airfoil_cuda:   airfoil_op.cpp airfoil_kernels_cu.o Makefile
                $(MPICPP) $(CPPFLAGS) airfoil_op.cpp airfoil_kernels_cu.o \
                $(CUDA_INC) $(OP2_INC) $(HDF5_INC) \
                $(OP2_LIB) $(CUDA_LIB) -lcudart -lop2_cuda -lop2_hdf5 \
                $(HDF5_LIB) -o airfoil_cuda

airfoil_kernels_cu.o:   airfoil_kernels.cu      \
                save_soln_kernel.cu save_soln.h \
                adt_calc_kernel.cu  adt_calc.h  \
                res_calc_kernel.cu  res_calc.h  \
                bres_calc_kernel.cu bres_calc.h \
                update_kernel.cu    update.h    \
                Makefile
                nvcc $(INC) $(NVCCFLAGS) $(OP2_INC) $(HDF5_INC) \
                -I /home/gihan/openmpi-intel/include \
                -c -o airfoil_kernels_cu.o airfoil_kernels.cu
```

Figure 14: CUDA with HDF5 build

On the other hand the functions facilitating MPI parallel file I/O with hdf5 are contained in the MPI back-end implicitly. Thus linking should not be done with `libop2_hdf5.a` in this case, for example:

```
HDF5_INC = -I$(HDF5_INSTALL_PATH)/include
HDF5_LIB = -L$(HDF5_INSTALL_PATH)/lib -lhdf5 -lz

airfoil_mpi_cuda: airfoil_op.cpp airfoil_kernels_mpi_cu.o Makefile
                  $(MPICPP) $(MPIFLAGS) airfoil_op.cpp \
                  -lm airfoil_kernels_mpi_cu.o \
                  $(OP2_INC) $(PARMETIS_INC) $(PTSCOTCH_INC) $(HDF5_INC) \
                  $(OP2_LIB) -lop2_mpi_cuda \
                  $(PARMETIS_LIB) $(PTSCOTCH_LIB) \
                  $(HDF5_LIB) $(CUDA_LIB) -lcudart -o airfoil_mpi_cuda

airfoil_kernels_mpi_cu.o: airfoil_kernels.cu \
                save_soln_kernel.cu  save_soln.h \
                adt_calc_kernel.cu   adt_calc.h  \
                res_calc_kernel.cu   res_calc.h  \
                bres_calc_kernel.cu  bres_calc.h \
                update_kernel.cu     update.h    \
                Makefile
                nvcc  $(INC) $(NVCCFLAGS) $(OP2_INC) \
                -I $(MPI_INSTALL_PATH)/include \
                -c -o airfoil_kernels_mpi_cu.o airfoil_kernels.cu
```

Figure 15: MPI+CUDA with HDF5 build

6 OP2 Example Application Directory Structure and Cmake Build Process

Airfoil is one of several example applications that is available with the public OP2 release. Currently there are three example applications: Airfoil, Aero and Jac[1]. The the C++ versions of these applications appear under `OP2-Common/apps/c` directory. Both Airfoil and Aero has been implemented using both hdf5 I/O and plain ASCI file I/O (as an example of user specified I/O). The `/sp` and `/dp` directories in each gives the single- and double-precision versions of the application. These versions have been used extensively in our published work for performance benchmarking of OP2.

Each individual application can be built by invoking `make` in the respective directory. There is also a cmake build process that will build all the applications (for all back-ends). Invoking ./cmake.local within in the `OP2-Common/apps/c` directory will build and install the applications in `OP2-Common/apps/c/bin`. More details are given in the README file.

References

[1] Giles, M. Hydra. http://people.maths.ox.ac.uk/gilesm/hydra.html.

[2] Giles, M. B., Duta, M. C., Muller, J. D., and Pierce, N. A. Algorithm Developments for Discrete Adjoint Methods. *AIAA Journal 42*, 2 (2003), 198–205.

[3] Giles, M. B., Mudalige, G. R., Sharif, Z., Markall, G., and Kelly, P. H. J. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev. 38*, 4 (March 2011), 9–15.

[4] Giles, M. B., Mudalige, G. R., Sharif, Z., Markall, G., and Kelly, P. H. J. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal 55*, 2 (2012), 168–180.

[5] Mudalige, G. R., Giles, M. B., Bertolli, C., and Kelly., P. H. J. Predictive modeling and analysis of OP2 on distributed memory GPU clusters. *SIGMETRICS Perform. Eval. Rev. 40*, 2 ((to appear)2012).

[6] Mudalige, G. R., Reguly, I., Giles, M. B., Bertolli, C., and Kelly., P. H. J. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Proceedings of Innovative Parallel Computing (InPar '12).* (San Jose, California, May 2012), IEEE.

[1] jac1 and jac2 implement the same application but jac2 differs in that it is intended to be a debugging application for the MATLAB generator