Tutorial: Migrating an application to use OPS

István Reguly

Pázmány Péter Catholic University, Faculty of Information Technology and Bionics

reguly.istvan@itk.ppke.hu

Apr 12, 2018

Overview

OPS Abstraction

- OPS is a Domain Specific Language embedded in C/C++ and Fortran, targeting multi-block structured mesh computations
- The abstraction has two distinct components: the definition of the mesh, and operations over the mesh
  - Defining a number of 1-3D [1] blocks, and on them a number of datasets, which have specific extents in the different dimensions
  - Describing a parallel loop over a given block, with a given iteration range, executing a given “kernel function” at each grid point, and describing what datasets are going to be accessed and how.
  - Additionally, one needs to declare stencils (access patterns) that will be used in parallel loops to access datasets, and any global constants (read-only global scope variables)
- Data and computations expressed this way can be automatically managed and parallelised by the OPS library.

[1] Higher dimensions supported in the backend, but not by the code generators yet

Example application

Our example application is a simple 2D iterative Laplace equation solver.

- Go to the `OPS/apps/c/laplace2d_tutorial/original` directory
- Open the `laplace2d.cpp` file
- It uses an $imax * jmax$ grid, with an additional 1 layers of boundary cells on all sides
- There are a number of loops that set the boundary conditions along the four edges
- The bulk of the simulation is spent in a `while` loop, repeating a stencil kernel with a maximum reduction, and a copy kernel
- Compile and run the code!

Original - Initialisation

```
//Size along y
int jmax = 4094;
//Size along x
int imax = 4094;

int iter_max = 100;
double pi  = 2.0 * asin(1.0);
const double tol = 1.0e-6;
double error     = 1.0;

double *A;
double *Anew;
double *y0;

A    = (double *)malloc((imax+2)*(jmax+2)*sizeof(double));
Anew = (double *)malloc((imax+2)*(jmax+2)*sizeof(double));
y0   = (double *)malloc((imax+2)*sizeof(double));

memset(A, 0, (imax+2)*(jmax+2)*sizeof(double));
```

Original - Boundary loops

```
// set boundary conditions
for (int i = 0; i < imax+2; i++)
  A[(0)*(imax+2)+i]     = 0.0;

for (int i = 0; i < imax+2; i++)
  A[(jmax+1)*(imax+2)+i] = 0.0;

for (int j = 0; j < jmax+2; j++)
{
  A[(j)*(imax+2)+0] = sin(pi * j / (jmax+1));
}

for (int j = 0; j < imax+2; j++)
{
  A[(j)*(imax+2)+imax+1] = sin(pi * j / (jmax+1))*exp(-pi);
}
```

- Note how in the latter two loops the loop index is used

Original - Main iteration

```
while ( error > tol && iter < iter_max ) {
    error = 0.0;
    for( int j = 1; j < jmax+1; j++ ) {
      for( int i = 1; i < imax+1; i++) {
        Anew[(j)*(imax+2)+i] = 0.25f * (
                A[(j)*(imax+2)+i+1] + A[(j)*(imax+2)+i-1]
              + A[(j-1)*(imax+2)+i] + A[(j+1)*(imax+2)+i]);
        error = fmax( error, fabs(Anew[(j)*(imax+2)+i]
                                  -A[(j)*(imax+2)+i]));
      }
    }
    for( int j = 1; j < jmax+1; j++ ) {
      for( int i = 1; i < imax+1; i++) {
        A[(j)*(imax+2)+i] = Anew[(j)*(imax+2)+i];
      }
    }
    if(iter % 10 == 0) printf("%5d, %0.6f\n", iter, error);
    iter++;
  }
```

Building OPS

- To build OPS, we need a C/C++ compiler at the minimum, and for advanced parallelisations we will need an MPI compiler, a CUDA compiler, and an OpenCL compiler. To support parallel file I/O we need the parallel HDF5 library as well. We do not require all of the above to build and use OPS (but don't be surprised if some build targets fail).
- The OPS Makefiles rely on a set of environment flags to be set appropriately:

```
export OPS_COMPILER=gnu #or intel, cray, pgi, clang
export OPS_INSTALL_PATH=~/OPS/ops
export MPI_INSTALL_PATH=/opt/openmpi #MPI root folder
export CUDA_INSTALL_PATH=/usr/local/cuda #CUDA root folder
export OPENCL_INSTALL_PATH=/usr/local/cuda #OpenCL root folder
export HDF5_INSTALL_PATH=/opt/hdf5 #HDF5 root folder
export NV_ARCH=Pascal #if you use GPUs, their generation
```

- Having set these, you can type `make` in the `ops/c` subdirectory

Preparing to use OPS - step 1

First off, we need to include the appropriate header files, then initialise OPS, and at the end finalise it.

- Define that this application is 2D, include the OPS header file, and create a header file where the outlined “user kernels” will live

```
#define OPS_2D
#include <ops_seq.h>
#include "laplace_kernels.h"
```

- Initialise and finalise OPS

```
  //Initialise the OPS library, passing runtime args, and
    setting diagnostics level to low (1)
  ops_init(argc, argv,1);
  ...
  ops_exit();
```

- By this point you need OPS set up - take a look at the Makefile in step1, and observer that the include and library paths are added, and we link against ops_seq.

OPS declarations - step 2

- Now we can declare a block, and the datasets

```
ops_block block = ops_decl_block(2, "my_grid");
int size[] = {imax, jmax};
int base[] = {0,0};
int d_m[] = {-1,-1};
int d_p[] = {1,1};
ops_dat d_A    = ops_decl_dat(block, 1, size, base,
                     d_m, d_p, A,    "double", "A");
ops_dat d_Anew = ops_decl_dat(block, 1, size, base,
                     d_m, d_p, Anew, "double", "Anew");
```

- datasets have a size (number of grid points in each dimension). There is padding for halos or boundaries in the positive (`d_p`) and negative directions (`d_m`); here we use a 1 thick boundary layer. Base index can be defined as it may be different from 0 (e.g. in Fortran).
- item these with a 0 base index and a 1 wide halo, these datasets can be indexed from -1 to $size + 1$

OPS declarations - step 2

- OPS supports gradual conversion of applications to its API, but in this case the described data sizes will need to match: the allocated memory and its extents need to be correctly described to OPS. In this case we have two $(imax + 2) * (jmax + 2)$ size arrays, and the total size in each dimension needs to match $size[i] + d\_p[i] - d\_m[i]$. This is only supported for the sequential and OpenMP backends
- If a NULL pointer is passed, OPS will allocate the data internally

OPS declarations - step 2

- We also need to declare the stencils that will be used - most loops use a simple 1-point stencil, and one uses a 5-point stencil

```
//Two stencils, a 1-point, and a 5-point
int s2d_00[] = {0,0};
ops_stencil S2D_00 =
    ops_decl_stencil(2,1,s2d_00,"0,0");
int s2d_5pt[] = {0,0, 1,0, -1,0, 0,1, 0,-1};
ops_stencil S2D_5pt =
    ops_decl_stencil(2,5,s2d_5pt,"5pt");
```

- Different names may be used for stencils in your code, but we suggest using some convention

First parallel loop - step 3

- It's time to convert the first loop to use OPS:

```
  for (int i = 0; i < imax+2; i++)
    A[(0)*(imax+2)+i]     = 0.0;
```

- This is a loop on the bottom boundary of the domain, which is at the -1 index for our dataset, therefore our iteration range will be over the entire domain, including halos in the X direction, and the bottom boundary in the Y direction. The iteration range is given as beginning (inclusive) and end (exclusive) indices in the x, y, etc. directions.

```
  int bottom_range[] = {-1, imax+1, -1, 0};
```

- Next, we need to outline the "user kernel" into `laplace_kernels.h`, and place the appropriate access macro - $OPS\_ACCN(i,j)$, where N is the index of the argument in the kernel's formal parameter list, and (i,j) are the stencil offsets in the X and Y directions respectively.

```
void set_zero(double *A) {
  A[OPS_ACC0(0,0)] = 0.0;
}
```

First parallel loop - step 3

- The OPS parallel loop can now be written as follows:

```
ops_par_loop(set_zero,"set_zero",block,2,bottom_range,
    ops_arg_dat(d_A, 1, S2D_00, "double", OPS_WRITE));
```

- The loop will execute `set_zero` at each grid point defined in the iteration range, and write the dataset `d_A` with the 1-point stencil
- The `ops_par_loop` implies that the order in which grid points will be executed will not affect the end result (within machine precision)

More parallel loops - step 3

- There are three more loops which set values to zero, they can be trivially replaced with the code above, only altering the iteration range.
- In the main `while` loop, the second simpler loop simply copies data from one array to another, this time on the interior of the domain:

```
int interior_range[] = {0,imax,0,jmax};
ops_par_loop(copy, "copy", block, 2, interior_range,
    ops_arg_dat(d_A,    1, S2D_00, "double", OPS_WRITE),
    ops_arg_dat(d_Anew, 1, S2D_00, "double", OPS_READ));
```

- And the corresponding outlined user kernel is as follows. Observe how for `Anew` the `OPS_ACC1` macro is used, as it is the second argument

```
void copy(double *A, const double *Anew) {
  A[OPS_ACC0(0,0)] = Anew[OPS_ACC1(0,0)];
}
```

Indexes and global constants - step 4

- There are two sets of boundary loops which use the loop variable j - this is a common technique to initialise data, such as coordinates ($x = i * dx$).
- OPS has a special argument `ops_arg_idx` which gives us a globally coherent (over MPI) iteration index - between the bounds supplied in the iteration range

```
int left_range[] = {-1, 0, -1, jmax+1};
ops_par_loop(left_bndcon, "left_bndcon", block, 2,
     left_range,
    ops_arg_dat(d_A, 1, S2D_00, "double", OPS_WRITE),
    ops_arg_idx());
```

- And the corresponding outlined user kernel is as follows. Observe the `idx` argument and the $+1$ offset due to the difference in indexing:

```
void left_bndcon(double *A, const int *idx) {
  A[OPS_ACC0(0,0)] = sin(pi * (idx[1]+1) / (jmax+1));
}
```

Indexes and global constants - step 4

- This kernel also uses two variables, `jmax` and `pi` that do not depend in the iteration index - they are iteration space invariant. OPS has two ways of supporting this:
    - Global scope constants, through `ops_decl_const`, as done in this case: we need to move the declaration of the `imax`, `jmax` and `pi` variables to global scope (outside of main), and call the OPS API:

```
//declare and define global constants
  ops_decl_const("imax",1,"int",&imax);
  ops_decl_const("jmax",1,"int",&jmax);
  ops_decl_const("pi",1,"double",&pi);
```

      These variables do not need to be passed in to the user kernel, they are accessible in all user kernels
    - The other option is to explicitly pass it to the user kernel with `ops_arg_gbl`: this is for scalars and small arrays that should not be in global scope.

Complex stencils and reductions - step 5

- There is only one loop left, which uses a 5 point stencil and a reduction. It can be outlined as usual, and for the stencil, we will use `S2D_pt5`
-
```
ops_par_loop(apply_stencil, "apply_stencil", block, 2,
    interior_range,
   ops_arg_dat(d_A,     1, S2D_5pt, "double", OPS_READ),
   ops_arg_dat(d_Anew, 1, S2D_00, "double", OPS_WRITE),
   ops_arg_reduce(h_err, 1, "double", OPS_MAX));
```

- And the corresponding outlined user kernel is as follows. Observe the stencil offsets used to access the adjacent 4 points:

```
void apply_stencil(const double *A, double *Anew, double
    *error) {
  Anew[OPS_ACC1(0,0)] = 0.25f * ( A[OPS_ACC0(1,0)]
      + A[OPS_ACC0(-1,0)]
      + A[OPS_ACC0(0,-1)] + A[OPS_ACC0(0,1)]);
  *error = fmax( *error, fabs(Anew[OPS_ACC1(0,0)]
                              -A[OPS_ACC0(0,0)]));
}
```

Complex stencils and reductions - step 5

- The loop also has a special argument for the reduction, `ops_arg_reduce`. As the first argument, it takes a reduction handle, which has to be defined separately:

-
  ```
  ops_reduction  h_err  =  ops_decl_reduction_handle(
                           sizeof(double), "double", "error");
  ```
- Reductions may be increment (`OPS_INC`), min (`OPS_MIN`) or max (`OPS_MAX`). The user kernel will have to perform the reduction operation, reducing the passed in value as well as the computed value
- The result of the reduction can be queried from the handle as follows:

```
ops_reduction_result(h_err, &error);
```

- Multiple parallel loops may use the same handle, and their results will be combined, until the result is queried by the user. Parallel loops that only have the reduction handle in common are semantically independent

Handing it all to OPS - step 6

- We have now successfully converted all computations on the grid to OPS parallel loops
- In order for OPS to manage data and parallelisations better, we should let OPS allocate the datasets - instead of passing in the pointers to memory allocated by us, we just pass in NULL (`A` and `Anew`)
    - Parallel I/O can be done using HDF5 - see the `ops_hdf5.h` header
- All data and parallelisation is now handed to OPS. We can now also compile the developer MPI version of the code - see the Makefile, and try building `laplace2d_mpi`

Code generation - step 7

- Now that the developer versions of our code work, it's time to generate code. On the console, type:

```
$OPS_INSTALL_PATH/../ops_translator/c/ops.py laplace2d.cpp
```

- We have provided a makefile which can use several different compilers (intel, cray, pgi, clang), we suggest modifying it for your own application
- Try building CUDA, OpenMP, MPI+CUDA, MPI+OpenMP, and other versions of the code
- You can take a look at the generated kernels for different parallelisations under the appropriate subfolders
- If you add the $-OPS\_DIAGS = 2$ runtime flag, at the end of execution, OPS will report timings and achieved bandwidth for each of your kernels
- For more, see the user guide...

Code generated versions

- OPS will generate and compile a large number of different versions
    - `laplace2d_dev_seq` and `laplace2d_dev_mpi`: these versions do not use code generation, they are intended for development only
    - `laplace2d_seq` and `laplace2d_mpi`: baseline sequential and MPI implementations
    - `laplace2d_openmp`: baseline OpenMP implementation
    - `laplace2d_cuda`, `laplace2d_opencl`, `laplace2d_openacc`: implementations targeting GPUs
    - `laplace2d_mpi_inline`: optimised implementation with MPI+OpenMP
    - `laplace2d_tiled`: optimised implementation with OpenMP that improves spatial and temporal locality

Optimisations

- Try the following performance tuning options
  - `laplace2d_cuda`, `laplace2d_opencl`: you can set the `OPS_BLOCK_SIZE_X` and `OPS_BLOCK_SIZE_Y` runtime arguments to control thread block or work group sizes
  - `laplace2d_mpi_cuda`, `laplace2d_mpi_openacc`: add the `-gpudirect` runtime flag to enable GPU Direct communications
  - `laplace2d_tiled`, `laplace2d_mpi_tiled`: add the `OPS_TILING` runtime flag, and move `-OPS_DIAGS=3`: see the cache blocking tiling at work. For some applications, such as this one, the initial guess gives too large tiles, try setting `OPS_CACHE_SIZE` to a lower value (in MB, for L3 size). Thread affinity control and using 1 process per socket is strongly recommended. E.g. `OMP_NUM_THREADS=20 numactl --cpunodebind=0 ./laplace2d_tiled -OPS_DIAGS=3 OPS_TILING OPS_CACHE_SIZE=5`. Over MPI, you will have to set `OPS_TILING_MAXDEPTH` to extend halo regions.

Optimisations - tiling

- Tiling uses lazy execution: as parallel loops follow one another, they are not executed, but put in a queue, and only once some data needs to be returned to the user (e.g. result of a reduction) do these loops have to be executed
- With a chain of loops queued, OPS can analyse them together and come up with a tiled execution schedule
- Works over MPI too: OPS extends the halo regions, and does one big halo exchange instead of several smaller ones
- In the current laplace2d code, every stencil application loop is also doing a reduction, therefore only two loops are queued. Try modifying the code so the reduction only happens every 10 iterations!
    - On A Xeon E5-2650, one can get a 2.5x speedup