

---

# **Oxford Parallel library for Structured mesh solvers**

*Release latest*

**Gihan Mudalige, Istvan Reguly, Mike Giles**

**May 16, 2024**



## CONTENTS:

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                | <b>1</b>  |
| 1.1      | Overview . . . . .                                 | 1         |
| 1.2      | Licencing . . . . .                                | 1         |
| 1.3      | Citing . . . . .                                   | 1         |
| 1.4      | Support . . . . .                                  | 2         |
| 1.5      | Funding . . . . .                                  | 2         |
| <b>2</b> | <b>Getting Started</b>                             | <b>3</b>  |
| 2.1      | Dependencies . . . . .                             | 3         |
| 2.2      | Obtaining OPS . . . . .                            | 4         |
| 2.3      | Build OPS . . . . .                                | 5         |
| <b>3</b> | <b>Developing an OPS Application</b>               | <b>7</b>  |
| 3.1      | OPS Abstraction . . . . .                          | 7         |
| 3.2      | Example Application . . . . .                      | 7         |
| 3.3      | Original - Initialisation . . . . .                | 8         |
| 3.4      | Original - Boundary loops . . . . .                | 8         |
| 3.5      | Original - Main iteration . . . . .                | 9         |
| 3.6      | Build OPS . . . . .                                | 9         |
| 3.7      | Step 1 - Preparing to use OPS . . . . .            | 9         |
| 3.8      | Step 2 - OPS declarations . . . . .                | 10        |
| 3.9      | Step 3 - First parallel loop . . . . .             | 10        |
| 3.10     | Step 4 - Indexes and global constants . . . . .    | 11        |
| 3.11     | Step 5 - Complex stencils and reductions . . . . . | 12        |
| 3.12     | Step 6 - Handing it all to OPS . . . . .           | 13        |
| 3.13     | Step 7 - Code generation . . . . .                 | 13        |
| 3.14     | Code generated versions . . . . .                  | 13        |
| 3.15     | Optimizations - general . . . . .                  | 13        |
| 3.16     | Optimizations - tiling . . . . .                   | 14        |
| <b>4</b> | <b>OPS API</b>                                     | <b>15</b> |
| 4.1      | Overview . . . . .                                 | 15        |
| 4.2      | Key Concepts and Structure . . . . .               | 15        |
| 4.3      | OPS C and C++ API . . . . .                        | 17        |
| 4.4      | Runtime Flags and Options . . . . .                | 35        |
| 4.5      | Doxygen . . . . .                                  | 35        |
| <b>5</b> | <b>Examples</b>                                    | <b>37</b> |
| <b>6</b> | <b>Performance Tuning</b>                          | <b>39</b> |
| 6.1      | Executing with GPUDirect . . . . .                 | 39        |

|          |                                 |           |
|----------|---------------------------------|-----------|
| 6.2      | Cache-blocking Tiling . . . . . | 39        |
| 6.3      | OpenMP and OpenMP+MPI . . . . . | 40        |
| 6.4      | CUDA arguments . . . . .        | 40        |
| 6.5      | OpenCL arguments . . . . .      | 40        |
| <b>7</b> | <b>Developer Guide</b>          | <b>41</b> |
| 7.1      | Contributing . . . . .          | 41        |
| <b>8</b> | <b>Publications</b>             | <b>43</b> |
| <b>9</b> | <b>Indices and tables</b>       | <b>45</b> |

## INTRODUCTION

### 1.1 Overview

OPS (Oxford Parallel library for Structured mesh solvers) is a high-level embedded domain specific language (eDSL) for writing **multi-block structured mesh** algorithms, and the corresponding software library and code translation tools to enable automatic parallelisation on multi-core and many-core architectures. Multi-block structured meshes consists of an unstructured collection of structured meshes. The OPS API is embedded in C/C++ and Fortran.

The current OPS eDSL supports generating code targeting multi-core/multi-threaded CPUs, many-core GPUs and clusters of CPUs and GPUs using a range of parallelization models including SIMD vectorization, OpenMP, CUDA, OpenCL, OpenACC and their combinations with MPI. There is also experimental support for parallelizations using SYCL and AMD HIP. Various optimizations for each parallelization can be generated automatically, including cache blocking tiling to improve locality. The OPS API and library can also be used to solve multi-dimensional tridiagonal systems using the [tridsolver](#) library.

These pages provide detailed documentation on using OPS, including an installation guide, developing and running OPS applications, the OPS API, developer documentation and performance tuning.

### 1.2 Licencing

OPS is released as an open-source project under the BSD 3-Clause License. See the [LICENSE](#) file for more information.

### 1.3 Citing

To cite OPS, please reference the following paper:

I. Z. Reguly, G. R. Mudalige and M. B. Giles, Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS, in IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 4, pp. 873-886, 1 April 2018, doi: 10.1109/TPDS.2017.2778161.

```
@ARTICLE{Reguly_et_al_2018,
  author={Reguly, István Z. and Mudalige, Gihan R. and Giles, Michael B.},
  journal={IEEE Transactions on Parallel and Distributed Systems},
  title={Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS},
  year={2018},
  volume={29},
  number={4},
  pages={873-886},
  doi={10.1109/TPDS.2017.2778161}}
```

Full list of publications from the OPS project can be found in the [Publications](#) section.

## 1.4 Support

The preferred method of reporting bugs and issues with OPS is to submit an issue via the repository's issue tracker. Users can also email the authors directly by contacting the [OP-DSL team](#).

## 1.5 Funding

The development of OPS was in part supported by the UK Engineering and Physical Sciences Research Council (EPSRC) grants [EP/K038494/1](#) ("Future-proof massively-parallel execution of multi-block applications"), [EP/J010553/1](#) ("Software for Emerging Architectures - ASEArch"), The UK Turbulence Consortium grant [EP/T026170/1](#), The Janos Bolyai Research Scholarship of the Hungarian Academy of Sciences, the Royal Society through their Industry Fellowship Scheme (INF/R1/180012), and the Thematic Research Cooperation Establishing Innovative Informatic and Information Solutions Project, which has been supported by the European Union and co-financed by the European Social Fund under grant number EFOP-3.6.2-16-2017-00013. Research funding support was also provided by the UK AWE under grants CDK0660 ("The Production of Predictive Models for Future Computing Requirements"), CDK0724 ("AWE Technical Outreach Programme"), AWE grant for "High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems" and the Numerical Algorithms Group [NAG](#).

Hardware resources for development and testing provided by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, the [ARCHER](#) and [ARCHER2](#) UK National Supercomputing Service, [University of Oxford Advanced Research Computing \(ARC\)](#) facility and through hardware donations and access provided by NVIDIA and Intel.

## GETTING STARTED

**Note:** The current CMakefile and relevant instructions are mainly tested on linux-based systems including Windows Subsystem for Linux

### 2.1 Dependencies

The following prerequisites and dependencies are required for building OPS. Building each of the **backends** are optional and depends on the hardware and/or capabilities you will be targeting.

#### CMake

CMake 3.18 or newer is required for using the CMake building system. If the latest version is not installed/shipped by default, it can be downloaded from <https://cmake.org/download/>, e.g., using the following script.

```
version=3.19.0
wget https://github.com/Kitware/CMake/releases/download/v$version/cmake-$version-Linux-
  ↪x86_64.sh
# Assume that CMake is going to be installed at /usr/local/cmake
cmake_dir=/usr/local/cmake
# sudo is not necessary for directories in user space.
sudo mkdir $cmake_dir
sudo sh ./cmake-$version-Linux-x86_64.sh --prefix=$cmake_dir --skip-license
sudo ln -s $cmake_dir/bin/cmake /usr/local/bin/cmake
```

#### Python

The Python dependencies (primarily used for the OPS code generator) are best installed by setting up a virtual environment so that required packages can be installed without superuser privileges. To set up the Python virtual environment and install the required dependant packages, ensure that you have Python3.9 or a more recent version with pip installed. Detailed instructions for installing virtual environment using pip can be found [here](#) Execute following **after cloning the OPS repository (see below)** to install required packages. Note OPS\_INSTALL\_PATH is the installation directory of OPS/ops:

```
#Install virtual environment using pip (if not installed earlier)
#Please set the OPS_INSTALL_PATH variable before running following commands
python3 -m pip install --user virtualenv

mkdir -p $OPS_INSTALL_PATH/../ops_translator/ops_venv
python3 -m venv $OPS_INSTALL_PATH/../ops_translator/ops_venv
source $OPS_INSTALL_PATH/../ops_translator/ops_venv/bin/activate
python3 -m pip install --upgrade pip
```

(continues on next page)

(continued from previous page)

```
python3 -m pip install -r $OPS_INSTALL_PATH/./ops_translator/requirements.txt
python3 -m pip install --force-reinstall libclang==16.0.6
```

These instructions can be executed by running the script `OPS/ops_translator/setup_venv.sh` file. After successfully setting up the Python virtual environment and installing the required dependent packages using the above instructions, you will need to activate the virtual environment by `source $OPS_INSTALL_PATH/./ops_translator/ops_venv/bin/activate` every time you want to use the code generator. Activating the virtual environment ensures that the code generator and its dependencies are isolated from the system-wide Python installation, avoiding conflicts and ensuring proper execution.

## HDF5

**HDF5** is required for parts of IO functionalities. The CMake build system **uses the parallel version by default** even for sequential codes, and automatically identify the library. If the automatic process fails, the path to the parallel HDF5 library can be specified by using `-DHDF5_ROOT`.

## CUDA Backend

The **CUDA** backend targets NVIDIA GPUs with a compute capability of 3.0 or greater. The CMake build system will detect the toolkit automatically. If the automatic process fails, the build system will compile the library without the CUDA support. Please use `-DCUDA_TOOLKIT_ROOT_DIR` to manually specify the path.

## HIP Backend

The HIP backend targets AMD GPUs and NVIDIA GPUs which are supported by HIP - either through its CUDA support or the **ROCm** stack (tested with `>=3.9`).

## SYCL Backend

The **SYCL** backend is currently in development and only working without MPI. It has been tested with Intel OneAPI (`>=2021.1`), Intel's public LLVM version, and hipSYCL (`>=0.9.1`), and runs on Intel CPUs and GPUs through Intel's OpenCL and Level Zero, NVIDIA and AMD GPUs both with the LLVM fork as well as hipSYCL. hipSYCL's OpenMP support covers most CPU architectures too.

## Tridiagonal Solver Backend

To use the tridiagonal solver OPS API in applications and build example applications such as `adi`, `adi_burger` and `adi_burger_3D` the open source tridiagonal solver (scalar) library needs to be cloned and built from the **Tridsolver repository**.

```
git clone https://github.com/OP-DSL/tridsolver.git
```

Details on building scalar tridiagonal solver library can be found in the **README** file located at the appropriate sub-directory.

## 2.2 Obtaining OPS

The latest OPS source code can be obtained by cloning the **OPS repository** using

```
git clone https://github.com/OP-DSL/OPS.git
```



## 2.3 Build OPS

### 2.3.1 Using cmake

#### Build library and example applications together

Create a build directory, and run CMake (version 3.18 or newer)

```
mkdir build
cd build
# Please see below for CMake options
cmake ${PATH_TO_OPS} -DBUILD_OPS_APPS=ON -DOPS_TEST=ON -DAPP_INSTALL_DIR=$HOME/OPS-APP -
↳DCMAKE_INSTALL_PREFIX=$HOME/OPS-INSTALL -DGPU_NUMBER=1
make # IEEE=1 enable IEEE flags in compiler
make install # sudo is needed if a directory like /usr/local/ is chosen.
```

After installation, the library and the python translator can be found at the directory specified by CMAKE\_INSTALL\_PREFIX, together with the executable files for applications at APP\_INSTALL\_DIR.

#### Build library and example applications separately

In this mode, the library can be firstly built and installed as

```
mkdir build
cd build
# Please see below for CMake options
cmake ${PATH_TO_OPS} -DCMAKE_INSTALL_PREFIX=$HOME/OPS-INSTALL
make # IEEE=1 enable IEEE flags in compiler
make install # sudo is needed if a system directory is chosen,
```

Then the application can be built as:

```
mkdir appbuild
cd appbuild
# Please see below for CMake options
cmake ${PATH_TO_APPS} -DOPS_INSTALL_DIR=$HOME/OPS-INSTALL -DOPS_TEST=ON -DAPP_INSTALL_
↳DIR=$HOME/OPS-APP -DGPU_NUMBER=1
make # IEEE=1 this option is important for applications to get accurate results
```

#### cmake options

- -DCMAKE\_BUILD\_TYPE=Release - enable optimizations
- -DBUILD\_OPS\_APPS=ON - build example applications (Library CMake only)
- -DOPS\_TEST=ON - enable the tests
- -DCMAKE\_INSTALL\_PREFIX= - specify the installation direction for the library (/usr/local by default, Library CMake only)
- -DAPP\_INSTALL\_DIR= - specify the installation direction for the applications (\$HOME/OPS-APPS by default)
- -DGPU\_NUMBER= - specify the number of GPUs used in the tests
- -DOPS\_INSTALL\_DIR= - specify where the OPS library is installed (Application CMake only, see [here](#))

- `-DOPS_VERBOSE_WARNING=ON` - show verbose output during building process

## 2.3.2 Using Makefiles

### Set up environmental variables:

- `OPS_COMPILER` - compiler to be used (Currently supports Intel, PGI and Cray compilers, but others can be easily incorporated by extending the Makefiles used in step 2 and 3)
- `OPS_INSTALL_PATH` - Installation directory of OPS/ops
- `CUDA_INSTALL_PATH` - Installation directory of CUDA, usually `/usr/local/cuda` (to build CUDA libs and applications)
- `OPENCL_INSTALL_PATH` - Installation directory of OpenCL, usually `/usr/local/cuda` for NVIDIA OpenCL implementation (to build OpenCL libs and applications)
- `MPI_INSTALL_PATH` - Installation directory of MPI (to build MPI based distributed memory libs and applications)
- `HDF5_INSTALL_PATH` - Installation directory of HDF5 (to support HDF5 based File I/O)

See example scripts (e.g. `source_intel`, `source_pgi_15.10`, `source_cray`) under `OPS/ops/scripts` that sets up the environment for building with various compilers (Intel, PGI, Cray).

### Build back-end library

For C/C++ back-end use Makefile under `OPS/ops/c` (modify Makefile if required). The libraries will be built in `OPS/ops/c/lib`

```
cd $OPS_INSTALL_PATH/c
make
```

For Fortran back-end use Makefile under `OPS/ops/fortran` (modify Makefile if required). The libraries will be built in `OPS/ops/fortran/lib`

```
cd $OPS_INSTALL_PATH/fortran
make
```

### Build example applications

For example to build `CloverLeaf_3D` under `OPS/apps/c/CloverLeaf_3D`

```
cd ../apps/c/Cloverleaf_3D/
make
```

## DEVELOPING AN OPS APPLICATION

This page provides a tutorial in the basics of using OPS for multi-block structured mesh application development. This is taken from a [presentation](#) given initially in April 2018 and subsequently updated for the latest release of OPS.

### 3.1 OPS Abstraction

OPS is a Domain Specific Language embedded in C/C++ and Fortran, targeting the development of 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 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 mesh 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. Higher dimensions are supported in the backend, but not currently by the code generators.

### 3.2 Example Application

In this tutorial we will use an example application, a simple 2D iterative Laplace equation solver.

- Go to the OPS/apps/c/laplace2dtutorial/original directory
- Open the `laplace2d.cpp` file
- It uses an  $imax \times jmax$  mesh, 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 !

Note: The following tutorial details the step-by-step approach for using OPS for Laplace (C version) application development. Similar step-by-step approach is also followed for the Laplace Fortran version and can be found at OPS/apps/fortran/laplace2dtutorial.

### 3.3 Original - Initialisation

The original code begins with initializing the data arrays used in the calculation:

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

### 3.4 Original - Boundary loops

The application sets 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.

## 3.5 Original - Main iteration

The main iterative loop is a while loop iterating until the error tolerance is at a set level and the number of iterations are less than the maximum set.

```
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++;
}
```

## 3.6 Build OPS

Build OPS using instructions in the [Getting Started](#) page.

## 3.7 Step 1 - Preparing to use OPS

Firstly, 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 “elemental kernels” will live.

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

- Initialise and finalise OPS

```
int main(int argc, const char** argv) {
    //Initialise the OPS library, passing runtime args, and setting diagnostics level to
    ↪ low (1)
    ops_init(argc, argv,1);
    ...
    ...
    //Finalising the OPS library
    ops_exit();
}
```

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

### 3.8 Step 2 - OPS declarations

Now declare a block and data on the block :

```
//The 2D block
ops_block block = ops_decl_block(2, "my_grid");

//The two datasets
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");
```

Data sets have a size (number of mesh points in each dimension). There is passing 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 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 example 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.

We also need to declare the stencils that will be used - in this example 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.

### 3.9 Step 3 - First parallel loop

You can now 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 “elemental” into `laplacekernels.h`, and place the appropriate access objects - `ACC<double> &A`, in the kernel’s formal parameter list, and `(i,j)` are the stencil offsets in the X and Y directions respectively:

```
void set_zero(ACC<double> &A) {
    A(0,0) = 0.0;
}
```

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 mesh 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 mesh points will be executed will not affect the end result (within machine precision).

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 elemental kernel is as follows:

```
void copy(ACC<double> &A, const ACC<double> &Anew) {
    A(0,0) = Anew(0,0);
}
```

## 3.10 Step 4 - Indexes and global constants

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 = idx`). OPS has a special argument `ops_arg_idx` which gives us a globally coherent (including over MPI) iteration index - between the bounds supplied in the iteration range.

```
ops_par_loop(left_bndcon, "left_bndcon", block, 2, left_range,
             ops_arg_dat(d_Anew, 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(ACC<double> &A, const int *idx) {
    A(0,0) = sin(pi * (idx[1]+1) / (jmax+1));
}
```

This kernel also uses two variables, `jmax` and `pi` that do not depend on the iteration index - they are iteration space invariant. OPS has two ways of supporting this:

1. Global scope constants, through `ops_decl_const`, as done in this example: 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 elemental kernel, they are accessible in all elemental kernels.

1. The other option is to explicitly pass it to the elemental kernel with `ops_arg_gbl`: this is for scalars and small arrays that should not be in global scope.

### 3.11 Step 5 - Complex stencils and reductions

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 `S2Dpt5`.

```
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 elemental kernel is as follows. Observe the stencil offsets used to access the adjacent 4 points:

```
void apply_stencil(const ACC<double> &A, ACC<double> &Anew, double *error) {
    Anew(0,0) = 0.25f * ( A(1,0) + A(-1,0)
                       + A(0,-1) + A(0,1));
    *error = fmax( *error, fabs(Anew(0,0)-A(0,0)));
}
```

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.



## 3.12 Step 6 - Handing it all to OPS

We have now successfully converted all computations on the mesh 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`.

## 3.13 Step 7 - Code generation

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

```
$OPSINSTALLPATH/../../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 applications. 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 options, see [Runtime Flags and Options](#).

## 3.14 Code generated versions

OPS will generate and compile a large number of different versions.

- `laplace2d_dev_seq` and `laplace2d_dev_mpi` : these 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_openc1`, `laplace2d_openacc` : implementations targeting GPUs
- `laplace2d_mpiinline` : optimised implementation with MPI+OpenMP
- `laplace2d_tiled`: optimised implementation with OpenMP that improves spatial and temporal locality

## 3.15 Optimizations - general

Try the following performance tuning options

- `laplace2d_cuda`, `laplace2d_openc1` : 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

## 3.16 Optimizations - 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.

This works over MPI as well: 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.

The following versions can be executed with the tiling optimizations.

- `laplace2d_tiled`, `laplace2d_mpi_tiled` : add the `OPS_TILING` runtime flag, and move `-OPSDIAGS=3` to 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 ./laplace2dtiled -OPSDIAGS=3 OPS_TILING OPS_CACHE_SIZE=5`. Over MPI, you will have to set `OPS_TILING_MAX_DEPTH` to extend halo regions.

## 4.1 Overview

The key characteristic of structured mesh applications is the implicit connectivity between neighboring mesh elements (such as vertices, cells). The main idea is that operations involve looping over a “rectangular” multi-dimensional set of mesh points using one or more “stencils” to access data. In multi-block meshes, we have several structured blocks. The connectivity between the faces of different blocks can be quite complex, and in particular they may not be oriented in the same way, i.e. an  $i, j$  face of one block may correspond to the  $j, k$  face of another block. This is awkward and hard to handle simply.

## 4.2 Key Concepts and Structure

The OPS API allows to declare a computation over such multi-block structured meshes. An OPS application can generally be declared in two key parts: (1) initialisation and (2) iteration over the mesh (carried out as a parallel loop). During the initialisation phase, one or more blocks (we call these `ops_blocks`) are defined: these only have a dimensionality (i.e. 1D, 2D, etc.), and serve to group datasets together. Datasets are defined on a block, and have a specific size (in each dimension of the block), which may be slightly different across different datasets (e.g. staggered grids), in some directions they may be degenerate (a size of 1), or they can represent data associated with different multigrid levels (where their size is a multiple or a fraction of other datasets). Datasets can be declared with empty (NULL) pointers, then OPS will allocate the appropriate amount of memory, may be passed non-NULL pointers (currently only supported in non-MPI environments), in which case OPS will assume the memory is large enough for the data and the block halo, and there are HDF5 dataset declaration routines which allow the distributed reading of datasets from HDF5 files. The concept of blocks is necessary to group datasets together, as in a multi-block problem, in a distributed memory environment, OPS needs to be able to determine how to decompose the problem.

The initialisation phase usually also consists of defining the stencils to be used later on (though they can be defined later as well), which describe the data access patterns used in parallel loops. Stencils are always relative to the “current” point; e.g. if at iteration  $(i, j)$ , we wish to access  $(i - 1, j)$  and  $(i, j)$ , then the stencil will have two points:  $\{(-1, 0), (0, 0)\}$ . To support degenerate datasets (where in one of the dimensions the dataset’s size is 1), as well as for multigrid, there are special strided, restriction, and prolongation stencils: they differ from normal stencils in that as one steps through a grid in a parallel loop, the stepping is done with a non-unit stride for these datasets. For example, in a 2D problem, if we have a degenerate dataset called `xcoords`, size  $(N, 1)$ , then we will need a stencil with stride  $(1, 0)$  to access it in a regular 2D loop.

Finally, the initialisation phase may declare a number of global constants - these are variables in global scope that can be accessed from within elemental kernels, without having to pass them in explicitly. These may be scalars or small arrays, generally for values that do not change during execution, though they may be updated during execution with repeated calls to `ops_decl_const`.

The initialisation phase is terminated by a call to `ops_partition`.

The bulk of the application consists of parallel loops, implemented using calls to `ops_par_loop`. These constructs work with datasets, passed through the opaque `ops_dat` handles declared during the initialisation phase. The iterations of parallel loops are semantically independent, and it is the responsibility of the user to enforce this: the order in which iterations are executed cannot affect the result (within the limits of floating point precision). Parallel loops are defined on a block, with a prescribed iteration range that is always defined from the perspective of the dataset written/modified (the sizes of datasets, particularly in multigrid situations, may be very different). Datasets are passed in using `ops_arg_dat`, and during execution, values at the current grid point will be passed to the user kernel. These values are passed wrapped in a templated `ACC<>` object (templated on the type of the data), whose parentheses operator is overloaded, which the user must use to specify the relative offset to access the grid point's neighbours (which accesses have to match the declared stencil). Datasets written may only be accessed with a one-point, zero-offset stencil (otherwise the parallel semantics may be violated).

Other than datasets, one can pass in read-only scalars or small arrays that are iteration space invariant with `ops_arg_gbl` (typically weights,  $\delta t$ , etc. which may be different in different loops). The current iteration index can also be passed in with `ops_arg_idx`, which will pass a globally consistent index to the user kernel (i.e. also under MPI).

Reductions in loops are done using the `ops_arg_reduce` argument, which takes a reduction handle as an argument. The result of the reduction can then be acquired using a separate call to `ops_reduction_result`. The semantics are the following: a reduction handle after it was declared is in an “uninitialised” state. The first time it is used as an argument to a loop, its type is determined (increment/min/max), and is initialised appropriately ( $0, \infty, -\infty$ ), and subsequent uses of the handle in parallel loops are combined together, up until the point, where the result is acquired using `ops_reduction_result`, which then sets it back to an uninitialised state. This also implies, that different parallel loops, which all use the same reduction handle, but are otherwise independent, are independent and their partial reduction results can be combined together associatively and commutatively.

OPS takes responsibility for all data, its movement and the execution of parallel loops. With different execution hardware and optimisations, this means OPS will **re-organise** data as well as execution (potentially across different loops), and therefore **data accesses or manipulation should only be done through the OPS API**. There is an external data access API that allows access to the data stored by OPS which in turn allows interfacing with external libraries.

This restriction is exploited by a lazy execution mechanism in OPS. The idea is that OPS API calls that do not return a result need not be executed immediately, rather queued, and once an API call requires returning some data, operations in the queue are executed, and the result is returned. This allows OPS to analyse and optimise operations in the queue together. This mechanism is fully automated by OPS, and is used with the various `_tiled` executables. For more information on how to use this mechanism for improving CPU performance, see Section on Tiling. Some API calls triggering the execution of queued operations include `ops_reduction_result`, and the functions in the data access API.

To further clarify some of the important issues encountered when designing the OPS API, we note here some needs connected with a 3D application:

- When looping over the interior with loop indices  $i, j, k$ , often there are 1D arrays which are referenced using just one of the indices.
- To implement boundary conditions, we often loop over a 2D face, accessing both the 3D dataset and data from a 2D dataset.
- To implement periodic boundary conditions using dummy “halo” points, we sometimes have to copy one plane of boundary data to another. e.g. if the first dimension has size  $I$  then we might copy the plane  $i = I - 2$  to plane  $i = 0$ , and plane  $i = 1$  to plane  $i = I - 1$ .
- In multigrid, we are working with two grids with one having twice as many points as the other in each direction. To handle this we require a stencil with a non-unit stride.
- In multi-block grids, we have several structured blocks. The connectivity between the faces of different blocks can be quite complex, and in particular they may not be oriented in the same way, i.e. an  $i, j$  face of one block may correspond to the  $j, k$  face of another block.

OPS handle all of these different requirements through stencil definitions.

## 4.3 OPS C and C++ API

Both C and C++ styles API are provided for utilizing the capabilities provided by the OPS library. They are essentially the same although there are minor differences in syntax. The C++ API is mainly designed for data abstraction, which therefore provides better data encapsulation and the support of multiple instances and threading (OpenMP currently). In the following both C style routines and C++ class and methods will be introduced according to their functionality with a notice (C) or (C++). If there is no such notice, the routine either applies to both or might not provided by the C++ API.

To enable the C++ API, a compiler directive `OPS_CPP_API` is required.

### 4.3.1 Initialisation and termination routines

#### C Style

##### `ops_init`

**`void ops_init(int argc, char** argv, int diags_level)`**

This routine must be called before all other OPS routines

| Arguments                | Description   |
|--------------------------|---|
| <code>argc, argv</code>  | the usual command line arguments  |
| <code>diags_level</code> | an integer which defines the level of debugging diagnostics and reporting to be performed |

Currently, higher `diags_level`s does the following checks

`diags_level = 1` : no diagnostics, default to achieve best runtime performance.

`diags_level > 1` : print block decomposition and `ops_par_loop` timing breakdown.

`diags_level > 4` : print intra-block halo buffer allocation feedback (for OPS internal development only)

`diags_level > 5` : check if intra-block halo MPI sends depth match MPI receives depth (for OPS internal development only)

##### `ops_exit`

**`void ops_exit()`**

This routine must be called last to cleanly terminate the OPS computation.

## C++ style

With the C++ style APIs, all data structures (block, data and stencils etc ) are encapsulated into a class `OPS_instance`. Thus, we can allocate multiple instances of `OPS_instance` by using the class constructor, for example,

```
// Allocate an instance
OPS_instance *instance = new OPS_instance(argc,argv,1,ss);
```

where the meaning of arguments are same to the C API, while the extra argument (i.e., `ss`) is for accepting the messages.

An explicit termination is not needed for the C++ API, although we need to “delete” the instance in if it is allocated through pointer, i.e.,

```
delete instance;
```

## 4.3.2 Declaration routines

### Block

#### `ops_decl_block (C)`

**`ops_block ops_decl_block(int dims, char *name)`**

This routine defines a structured grid block.

| Arguments         | Description                        |
|-------------------|------------------------------------|
| <code>dims</code> | dimension of the block             |
| <code>name</code> | a name used for output diagnostics |

#### `OPS_instance::decl_block (C++)`

A method of the `OPS_instance` class for declaring a block, which accepts same arguments with the C style function. A `OPS_instance` object should be constructed before this. The method returns a pointer to a `ops_block` type variable, where `ops_block` is an alias to a pointer type of `ops_block_core`. An example is

```
ops_block grid2D = instance->decl_block(2, "grid2D");
```

#### `ops_decl_block_hdf5 (C)`

**`ops_block ops_decl_block_hdf5(int dims, char *name, char *file)`**

This routine reads the details of a structured grid block from a named HDF5 file

| Arguments         | Description   |
|-------------------|---|
| <code>dims</code> | dimension of the block                                  |
| <code>name</code> | a name used for output diagnostics                      |
| <code>file</code> | hdf5 file to read and obtain the block information from |

Although this routine does not read in any extra information about the block from the named HDF5 file than what is already specified in the arguments, it is included here for error checking (e.g. check if blocks defined in an HDF5 file is matching with the declared arguments in an application) and completeness.

**Dat (ops\_cat\_core)****ops\_decl\_dat (C)**

**ops\_dat ops\_decl\_dat(ops\_block block, int dim, int \*size, int \*base, int \*dm, int \*dp, T \*data, char \*type, char \*name)**

This routine defines a dataset.

| Arguments | Description  |
|-----------|--|
| block     | structured block   |
| dim       | dimension of dataset (number of items per grid element)                                  |
| size      | size in each dimension of the block  |
| base      | base indices in each dimension of the block  |
| d_m       | padding from the face in the negative direction for each dimension (used for block halo) |
| d_p       | padding from the face in the positive direction for each dimension (used for block halo) |
| data      | input data of type <i>T</i>  |
| type      | the name of type used for output diagnostics (e.g. double,float)                         |
| name      | a name used for output diagnostics   |

The *size* allows to declare different sized data arrays on a given block. *d\_m* and *d\_p* are depth of the “block halos” that are used to indicate the offset from the edge of a block (in both the negative and positive directions of each dimension).

**ops\_block\_core::decl\_dat (C++)**

The method `ops_block_core::decl_dat` is used to define a `ops_dat` object, which accepts almost same arguments with the C counterpart where the block argument is not necessary, e.g.,

```
//declare ops_dat with dim = 2
ops_dat dat0 = grid2D->decl_dat(2, size, base, d_m, d_p, temp, "double", "dat0");
ops_dat dat1 = grid2D->decl_dat(2, size, base, d_m, d_p, temp, "double", "dat1");
```

where `grid2D` is a `ops_block_core` object which shall be defined before this.

**ops\_decl\_dat\_hdf5 (C)**

**ops\_dat ops\_decl\_dat\_hdf5(ops\_block block, int dim, char \*type, char \*name, char \*file)**

This routine defines a dataset to be read in from a named hdf5 file

| Arguments | Description  |
|-----------|--|
| block     | structured block   |
| dim       | dimension of dataset (number of items per grid element)          |
| type      | the name of type used for output diagnostics (e.g. double,float) |
| name      | name of the dat used for output diagnostics                      |
| file      | hdf5 file to read and obtain the data from                       |

## Global constant

### ops\_decl\_const (C)

**void ops\_decl\_const(char const \* name, int dim, char const \* type, T \* data )**

This routine defines a global constant: a variable in global scope. Global constants need to be declared upfront so that they can be correctly handled for different parallelization. For e.g CUDA on GPUs. Once defined they remain unchanged throughout the program, unless changed by a call to ops\_update\_const(..). The name ' ' and type " parameters **must** be string literals since they are used in the code generation step

| Arguments | Description   |
|-----------|---|
| name      | a name used to identify the constant                              |
| dim       | dimension of dataset (number of items per element)                |
| type      | the name of type used for output diagnostics (e.g. double, float) |
| data      | pointer to input data of type <i>T</i>                            |

### OPS\_instance::decl\_const (C++)

The method accepts same arguments with its C counterpart.

## Halo definition

### ops\_decl\_halo (C)

**ops\_halo ops\_decl\_halo(ops\_dat from, ops\_dat to, int iter\_size, int from\_base, int \*to\_base, int \*from\_dir, int \*to\_dir)**

| Arguments | Description   |
|-----------|---|
| from      | origin dataset  |
| to        | destination dataset   |
| item_size | defines an iteration size (number of indices to iterate over in each direction) |
| from_base | indices of starting point in "from" dataset                                     |
| to_base   | indices of starting point in "to" dataset                                       |
| from_dir  | direction of incrementing for "from" for each dimension of iter_size            |
| to_dir    | direction of incrementing for "to" for each dimension of iter_size              |

A from\_dir [1,2] and a to\_dir [2,1] means that x in the first block goes to y in the second block, and y in first block goes to x in second block. A negative sign indicates that the axis is flipped. (Simple example: a transfer from (1:2,0:99,0:99) to (-1:0,0:99,0:99) would use iter\_size = [2,100,100], from\_base = [1,0,0], to\_base = [-1,0,0], from\_dir = [0,1,2], to\_dir = [0,1,2]. In more complex case this allows for transfers between blocks with different orientations.)



**OPS\_instance::decl\_halo (C++)**

The method accepts same arguments with its C counterpart.

**ops\_decl\_halo\_hdf5 (C)**

**ops\_halo ops\_decl\_halo\_hdf5(ops\_dat from, ops\_dat to, char\* file)**

This routine reads in a halo relationship between two datasets defined on two different blocks from a named HDF5 file

| Arguments | Description                                |
|-----------|--|
| from      | origin dataset                             |
| to        | destination dataset                        |
| file      | hdf5 file to read and obtain the data from |

**ops\_decl\_halo\_group (C)**

**ops\_halo\_group ops\_decl\_halo\_group(int nhalos, ops\_halo \*halos)**

This routine defines a collection of halos. Semantically, when an exchange is triggered for all halos in a group, there is no order defined in which they are carried out.

| Arguments | Description                     |
|-----------|---------------------------------|
| nhalos    | number of halos in <i>halos</i> |
| halos     | array of halos                  |

**OPS\_instance::decl\_halo\_group (C++)**

The method accepts same arguments with its C counterpart.

**Reduction handle****ops\_decl\_reduction\_handle (C)**

**ops\_reduction ops\_decl\_reduction\_handle(int size, char \*type, char \*name)** This routine defines a reduction handle to be used in a parallel loop

| Arguments | Description   |
|-----------|---|
| size      | size of data in bytes   |
| type      | the name of type used for output diagnostics (e.g. <code>double</code> , <code>float</code> ) |
| name      | name of the dat used for output diagnostics   |

\_\_{ void ops\_reduction\_result(ops\_reduction handle, T \*result) { This routine returns the reduced value held by a reduction handle. When OPS uses lazy execution, this will trigger the execution of all previously queued OPS operations. }

|handle| the *ops\_reduction* handle |result| a pointer to write the results to, memory size has to match the declared |

**OPS\_instance::decl\_reduction\_handle (C++)**

The method accepts same arguments with its C counterpart.

**Partition****ops\_partition (C)****ops\_partition(char \*method)**

Triggers a multi-block partitioning across a distributed memory set of processes. (links to a dummy function for single node parallelizations). This routine should only be called after all the ops\_halo ops\_decl\_block and ops\_halo ops\_decl\_dat statements have been declared

| Argu-ments | Description   |
|------------|---|
| method     | string describing the partitioning method. Currently this string is not used internally, but is simply a place-holder to indicate different partitioning methods in the future. |

**OPS\_instance::partition (C++)**

The method accepts same arguments with its C counterpart.

**4.3.3 Diagnostic and output routines****ops\_diagnostic\_output (C)****void ops\_diagnostic\_output()**

This routine prints out various useful bits of diagnostic info about sets, mappings and datasets. Usually used right after an ops\_partition() call to print out the details of the decomposition

**OPS\_instance::diagnostic\_output (C++)**

Same to the C counterpart.

**ops\_printf****void ops\_printf(const char \* format, ...)**

This routine simply prints a variable number of arguments; it is created is in place of the standard C printf function which would print the same on each MPI process

### ops\_timers

**void ops\_timers(double \*cpu, double \*et)** gettimeofday() based timer to start/end timing blocks of code

| Arguments | Description   |
|-----------|---|
| cpu       | variable to hold the CPU time at the time of invocation     |
| et        | variable to hold the elapsed time at the time of invocation |

### ops\_fetch\_block\_hdf5\_file

**void ops\_fetch\_block\_hdf5\_file(ops\_block block, char \*file)**

Write the details of an ops\_block to a named HDF5 file. Can be used over MPI (puts the data in an ops\_dat into an HDF5 file using MPI I/O)

| Arguments | Description             |
|-----------|-------------------------|
| block     | ops_block to be written |
| file      | hdf5 file to write to   |

### ops\_fetch\_stencil\_hdf5\_file

**void ops\_fetch\_stencil\_hdf5\_file(ops\_stencil stencil, char \*file)**

Write the details of an ops\_block to a named HDF5 file. Can be used over MPI (puts the data in an ops\_dat into an HDF5 file using MPI I/O)

| Arguments | Description               |
|-----------|---------------------------|
| stencil   | ops_stencil to be written |
| file      | hdf5 file to write to     |

### ops\_fetch\_dat\_hdf5\_file

**void ops\_fetch\_dat\_hdf5\_file(ops\_dat dat, const char \*file)**

Write the details of an ops\_block to a named HDF5 file. Can be used over MPI (puts the data in an ops\_dat into an HDF5 file using MPI I/O)

| Arguments | Description           |
|-----------|-----------------------|
| dat       | ops_dat to be written |
| file      | hdf5 file to write to |

### ops\_print\_dat\_to\_txtfile

**void ops\_print\_dat\_to\_txtfile(ops\_dat dat, char \*file)** Write the details of an ops\_block to a named text file. When used under an MPI parallelization each MPI process will write its own data set separately to the text file. As such it does not use MPI I/O. The data can be viewed using a simple text editor

| Arguments | Description              |
|-----------|--------------------------|
| dat       | ops_dat to to be written |
| file      | text file to write to    |

### ops\_timing\_output

**void ops\_timing\_output(FILE \*os)**

Print OPS performance performance details to output stream

| Arguments | Description  |
|-----------|--|
| os        | output stream, use stdout to print to standard out |

### ops\_NaNcheck

**void ops\_NaNcheck(ops\_dat dat)**

Check if any of the values held in the *dat* is a NaN. If a NaN is found, prints an error message and exits.

| Arguments | Description              |
|-----------|--------------------------|
| dat       | ops_dat to to be checked |

## 4.3.4 Halo exchange

### ops\_halo\_transfer (C)

**void ops\_halo\_transfer(ops\_halo\_group group)**

This routine exchanges all halos in a halo group and will block execution of subsequent computations that depend on the exchanged data.

| Arguments | Description    |
|-----------|----------------|
| group     | the halo group |

## 4.3.5 Parallel loop syntax

A parallel loop with N arguments has the following syntax:

### ops\_par\_loop

**void ops\_par\_loop(void (\*kernel)(...),char \*name, ops\_block block, int dims, int \*range, ops\_arg arg1,ops\_arg arg2, ..., ops\_arg argN )**

| Arguments | Description  |
|-----------|--|
| kernel    | user's kernel function with N arguments              |
| name      | name of kernel function, used for output diagnostics |
| block     | the ops_block over which this loop executes          |
| dims      | dimension of loop iteration                          |
| range     | iteration range array                                |
| args      | arguments  |

The **ops\_arg** arguments in **ops\_par\_loop** are provided by one of the following routines, one for global constants and reductions, and the other for OPS datasets.

### ops\_arg\_gbl

**ops\_arg ops\_arg\_gbl(T \*data, int dim, char \*type, ops\_access acc)**

Passes a scalar or small array that is invariant of the iteration space (not to be confused with ops\_decl\_const, which facilitates global scope variables).

| Arguments | Description                                       |
|-----------|---|
| data      | data array  |
| dim       | array dimension                                   |
| type      | string representing the type of data held in data |
| acc       | access type                                       |

### ops\_arg\_reduce

**ops\_arg ops\_arg\_reduce(ops\_reduction handle, int dim, char \*type, ops\_access acc)**

Passes a pointer to a variable that needs to be incremented (or swapped for min/max reduction) by the user kernel.

| Arguments | Description                                       |
|-----------|---|
| handle    | an <i>ops_reduction</i> handle                    |
| dim       | array dimension (according to <i>type</i> )       |
| type      | string representing the type of data held in data |
| acc       | access type                                       |

### ops\_arg\_dat

**ops\_arg ops\_arg\_dat(ops\_dat dat, ops\_stencil stencil, char \*type,ops\_access acc)**

Passes a pointer wrapped in an ACC<> object to the value(s) at the current grid point to the user kernel. The ACC object's parentheses operator has to be used for dereferencing the pointer.

| Arguments | Description  |
|-----------|--|
| dat       | dataset  |
| stencil   | stencil for accessing data                           |
| type      | string representing the type of data held in dataset |
| acc       | access type  |

### **ops\_arg\_idx**

#### **ops\_arg ops\_arg\_idx()**

Give you an array of integers (in the user kernel) that have the index of the current grid point, i.e. `idx[0]` is the index in x, `idx[1]` is the index in y, etc. This is a globally consistent index, so even if the block is distributed across different MPI partitions, it gives you the same indexes. Generally used to generate initial geometry.

## **4.3.6 Stencils**

The final ingredient is the stencil specification, for which we have two versions: simple and strided.

### **ops\_decl\_stencil (C)**

**ops\_stencil ops\_decl\_stencil(int dims,int points, int \*stencil, char \*name)**

| Arguments | Description                                 |
|-----------|---|
| dims      | dimension of loop iteration                 |
| points    | number of points in the stencil             |
| stencil   | stencil for accessing data                  |
| name      | string representing the name of the stencil |

### **OPS\_instance::decl\_stencil (C++)**

The method accepts same arguments with its C counterpart.

### **ops\_decl\_strided\_stencil (C)**

**ops\_stencil ops\_decl\_strided\_stencil(int dims, int points, int \*stencil, int \*stride, char \*name)**

| Arguments | Description                                 |
|-----------|---|
| dims      | dimension of loop iteration                 |
| points    | number of points in the stencil             |
| stencil   | stencil for accessing data                  |
| stride    | stride for accessing data                   |
| name      | string representing the name of the stencil |

### OPS\_instance::decl\_strided\_stencil (C++)

The method accepts same arguments with its C counterpart.

#### ops\_decl\_stencil\_hdf5

**ops\_stencil ops\_decl\_stencil\_hdf5(int dims,int points, char *name*, char *file*)**

| Arguments | Description                                 |
|-----------|---|
| dims      | dimension of loop iteration                 |
| points    | number of points in the stencil             |
| name      | string representing the name of the stencil |
| file      | hdf5 file to write to                       |

In the strided case, the semantics for the index of data to be accessed, for stencil point  $p$ , in dimension  $m$  are defined as

$$\text{stride}[m] * \text{loop\_index}[m] + \text{stencil}[p * \text{dims} + m]$$

where `loop_index[m]` is the iteration index (within the user-defined iteration space) in the different dimensions.

If, for one or more dimensions, both `stride[m]` and `stencil[p*dims+m]` are zero, then one of the following must be true;

- the dataset being referenced has size 1 for these dimensions
- these dimensions are to be omitted and so the dataset has dimension equal to the number of remaining dimensions.

See `OPS/apps/c/CloverLeaf/build_field.cpp` and `OPS/apps/c/CloverLeaf/generate.cpp` for an example `ops_decl_strided_stencil` declaration and its use in a loop, respectively.

These two stencil definitions probably take care of all of the cases in the Introduction except for multiblock applications with interfaces with different orientations – this will need a third, even more general, stencil specification. The strided stencil will handle both multigrid (with a stride of 2 for example) and the boundary condition and reduced dimension applications (with a stride of 0 for the relevant dimensions).

### 4.3.7 Checkpointing

OPS supports the automatic checkpointing of applications. Using the API below, the user specifies the file name for the checkpoint and an average time interval between checkpoints, OPS will then automatically save all necessary information periodically that is required to fast-forward to the last checkpoint if a crash occurred. Currently, when re-launching after a crash, the same number of MPI processes have to be used. To enable checkpointing mode, the `OPS_CHECKPOINT` runtime argument has to be used. (**Do we also need to define the CHECKPOINTING compiler directive?**)

#### ops\_checkpointing\_init

**bool ops\_checkpointing\_init(const char \*filename, double interval, int options)**

Initialises the checkpointing system, has to be called after `ops_partition`. Returns true if the application launches in restore mode, false otherwise.

| Arguments | Description   |
|-----------|---|
| filename  | name of the file for checkpointing. In MPI, this will automatically be post-fixed with the rank ID. |
| interval  | average time (seconds) between checkpoints  |
| options   | a combinations of flags, listed in <i>ops_checkpointing.h</i> , also see below                      |

- OPS\_CHECKPOINT\_INITPHASE - indicates that there are a number of parallel loops at the very beginning of the simulations which should be excluded from any checkpoint; mainly because they initialise datasets that do not change during the main body of the execution. During restore mode these loops are executed as usual. An example would be the computation of the mesh geometry, which can be excluded from the checkpoint if it is re-computed when recovering and restoring a checkpoint. The API call *void ops\_checkpointing\_initphase\_done()* indicates the end of this initial phase.
- OPS\_CHECKPOINT\_MANUAL\_DATLIST - Indicates that the user manually controls the location of the checkpoint, and explicitly specifies the list of *ops\_dats* to be saved.
- OPS\_CHECKPOINT\_FASTFW - Indicates that the user manually controls the location of the checkpoint, and it also enables fast-forwarding, by skipping the execution of the application (even though none of the parallel loops would actually execute, there may be significant work outside of those) up to the checkpoint
- OPS\_CHECKPOINT\_MANUAL - Indicates that when the corresponding API function is called, the checkpoint should be created. Assumes the presence of the above two options as well.

### ops\_checkpointing\_manual\_datlist

**void ops\_checkpointing\_manual\_datlist(int ndats, ops\_dat \*datlist)**

A user can call this routine at a point in the code to mark the location of a checkpoint. At this point, the list of datasets specified will be saved. The validity of what is saved is not checked by the checkpointing algorithm assuming that the user knows what data sets to be saved for full recovery. This routine should be called frequently (compared to check-pointing frequency) and it will trigger the creation of the checkpoint the first time it is called after the timeout occurs.

| Arguments | Description                                  |
|-----------|--|
| ndats     | number of datasets to be saved               |
| datlist   | arrays of <i>ops_dat</i> handles to be saved |

### ops\_checkpointing\_fastfw

**bool ops\_checkpointing\_fastfw(int nbytes, char \*payload)**

A use can call this routine at a point in the code to mark the location of a checkpoint. At this point, the specified payload (e.g. iteration count, simulation time, etc.) along with the necessary datasets, as determined by the checkpointing algorithm will be saved. This routine should be called frequently (compared to checkpointing frequency), will trigger the creation of the checkpoint the first time it is called after the timeout occurs. In restore mode, will restore all datasets the first time it is called, and returns true indicating that the saved payload is returned in payload. Does not save reduction data.

| Arguments | Description  |
|-----------|--|
| nbytes    | size of the payload in bytes                       |
| payload   | pointer to memory into which the payload is packed |



**ops\_checkpointing\_manual\_datlist\_fastfw**

**bool ops\_checkpointing\_manual\_datlist\_fastfw(int ndats, op\_dat \*datlist, int nbytes, char \*payload)**

Combines the manual datlist and fastfw calls.

| Arguments | Description  |
|-----------|--|
| ndats     | number of datasets to be saved                     |
| datlist   | arrays of <i>ops_dat</i> handles to be saved       |
| nbytes    | size of the payload in bytes                       |
| payload   | pointer to memory into which the payload is packed |

**ops\_checkpointing\_manual\_datlist\_fastfw\_trigger**

**bool ops\_checkpointing\_manual\_datlist\_fastfw\_trigger(int ndats, opa\_dat \*datlist, int nbytes, char \*payload)**

With this routine it is possible to manually trigger checkpointing, instead of relying on the timeout process. as such it combines the manual datlist and fastfw calls, and triggers the creation of a checkpoint when called.

| Arguments | Description  |
|-----------|--|
| ndats     | number of datasets to be saved                     |
| datlist   | arrays of <i>ops_dat</i> handles to be saved       |
| nbytes    | size of the payload in bytes                       |
| payload   | pointer to memory into which the payload is packed |

The suggested use of these **manual** functions is of course when the optimal location for checkpointing is known - one of the ways to determine that is to use the built-in algorithm. More details of this will be reported in a tech-report on checkpointing, to be published later.

**4.3.8 Access to OPS data**

This section describes APIs that give the user access to internal data structures in OPS and return data to user-space. These should be used cautiously and sparsely, as they can affect performance significantly

**ops\_dat\_get\_local\_npartitions (C)**

**int ops\_dat\_get\_local\_npartitions(ops\_dat dat)**

This routine returns the number of chunks of the given dataset held by the current process.

| Arguments | Description |
|-----------|-------------|
| dat       | the dataset |

**ops\_dat\_core::get\_local\_npartitions (C++)**

The C++ version of ops\_dat\_get\_local\_npartitions, which does not require input.

**ops\_dat\_get\_global\_npartitions (C)**

**int ops\_dat\_get\_global\_npartitions(ops\_dat dat)**

This routine returns the number of chunks of the given dataset held by all processes.

| Arguments | Description |
|-----------|-------------|
| dat       | the dataset |

**ops\_dat\_core::get\_global\_npartitions (C++)**

The C++ version of ops\_dat\_get\_global\_npartitions, which does not require input.

**ops\_dat\_get\_extents (C)**

**void ops\_dat\_get\_extents(ops\_dat dat, int part, int \*disp, int \*sizes)**

This routine returns the MPI displacement and size of a given chunk of the given dataset on the current process.

| Arguments | Description   |
|-----------|---|
| dat       | the dataset   |
| part      | the chunk index (has to be 0)   |
| disp      | an array populated with the displacement of the chunk within the ``global'' distributed array |
| sizes     | an array populated with the spatial extents   |

**ops\_dat\_core::get\_extents (C++)**

The C++ version of ops\_dat\_get\_extents where the arguments are the same except no need of the ops\_dat arguments.

**ops\_dat\_get\_raw\_metadata (C)**

**char\* ops\_dat\_get\_raw\_metadata(ops\_dat dat, int part, int \*disp, int \*size, int \*stride, int \*d\_m, int \*d\_p)**

This routine returns array shape metadata corresponding to the ops\_dat. Any of the arguments that are not of interest, may be NULL.

| Arguments | Description  |
|-----------|--|
| dat       | the dataset  |
| part      | the chunk index (has to be 0)  |
| disp      | an array populated with the displacement of the chunk within the ``global'' distributed array      |
| size      | an array populated with the spatial extents  |
| stride    | an array populated strides in spatial dimensions needed for column-major indexing                  |
| d_m       | an array populated with padding on the left in each dimension. Note that these are negative values |
| d_p       | an array populated with padding on the right in each dimension                                     |

**ops\_dat\_core::get\_raw\_metadata (C++)**

The C++ version of `ops_dat_get_raw_metadata` where the arguments are the same except no need of the `ops_dat` arguments.

**ops\_dat\_get\_raw\_pointer (C)**

**char\* ops\_dat\_get\_raw\_pointer(ops\_dat dat, int part, ops\_stencil stencil, ops\_memspace \*memspace)**

This routine returns a pointer to the internally stored data, with MPI halo regions automatically updated as required by the supplied stencil. The strides required to index into the dataset are also given.

| Argu-ments | Description  |
|------------|--|
| dat        | the dataset  |
| part       | the chunk index (has to be 0)  |
| stencil    | a stencil used to determine required MPI halo exchange depths  |
| memspace   | when set to OPS_HOST or OPS_DEVICE, returns a pointer to data in that memory space, otherwise must be set to 0, and returns whether data is in the host or on the device |

**ops\_dat\_core::get\_raw\_\_pointer (C++)**

The C++ version of `ops_dat_get_raw_pointer` where the arguments are the same except no need of the `ops_dat` arguments.

**ops\_dat\_release\_raw\_data (C)**

**void ops\_dat\_release\_raw\_data(ops\_dat dat, int part, ops\_access acc)**

Indicates to OPS that a dataset previously accessed with `ops_dat_get_raw_pointer` is released by the user, and also tells OPS how it was accessed.

A single call to `ops_dat_release_raw_data()` releases all pointers obtained by previous calls to `ops_dat_get_raw_pointer()` calls on the same `dat` and with the same `*memspace` argument, i.e. calls do not nest.

| Argu-ments | Description   |
|------------|---|
| dat        | the dataset   |
| part       | the chunk index (has to be 0)   |
| acc        | the kind of access that was used by the user (OPS_READ if it was read only, OPS_WRITE if it was overwritten, OPS_RW if it was read and written) |

**ops\_dat\_core::release\_raw\_data (C++)**

The C++ version of ops\_dat\_release\_raw\_data where the arguments are the same except no need of the ops\_dat arguments.

**ops\_dat\_fetch\_data (C)**

**void ops\_dat\_fetch\_data(ops\_dat dat, int part, int \*data)**

This routine copies the data held by OPS to the user-specified memory location, which needs to be at least as large as indicated by the sizes parameter of ops\_dat\_get\_extents.

| Arguments | Description                                     |
|-----------|---|
| dat       | the dataset                                     |
| part      | the chunk index (has to be 0)                   |
| data      | pointer to memory which should be filled by OPS |

**ops\_dat\_fetch\_data\_memspace (C)**

**void ops\_dat\_fetch\_data\_memspace(ops\_dat dat, int part, char \*data, ops\_memspace memspace)**

This routine copies the data held by OPS to the user-specified memory location, as which needs to be at least as large as indicated by the sizes parameter of ops\_dat\_get\_extents.

| Arguments | Description                                     |
|-----------|---|
| dat       | the dataset                                     |
| part      | the chunk index (has to be 0)                   |
| data      | pointer to memory which should be filled by OPS |
| memspace  | the memory space where the data pointer is      |

**ops\_dat\_core::fetch\_data (C++)**

The C++ version of ops\_dat\_fetch\_data\_memspace where the arguments the same except no need of the ops\_dat arguments.

**ops\_dat\_set\_data (C)**

**void ops\_dat\_set\_data(ops\_dat dat, int part, int \*data)**

This routine copies the data given by the user to the internal data structure used by OPS. User data needs to be laid out in column-major order and strided as indicated by the sizes parameter of ops\_dat\_get\_extents.

| Arguments | Description                                     |
|-----------|---|
| dat       | the dataset                                     |
| part      | the chunk index (has to be 0)                   |
| data      | pointer to memory which should be copied to OPS |

### ops\_dat\_set\_data\_memspace (C)

**void ops\_dat\_set\_data\_memspace(ops\_dat dat, int part, char \*data, ops\_memspace memspace)**

This routine copies the data given by the user to the internal data structure used by OPS. User data needs to be laid out in column-major order and strided as indicated by the sizes parameter of ops\_dat\_get\_extents.

| Arguments | Description                                     |
|-----------|---|
| dat       | the dataset                                     |
| part      | the chunk index (has to be 0)                   |
| data      | pointer to memory which should be copied to OPS |
| memspace  | the memory space where the data pointer is      |

### ops\_dat\_core::set\_data (C++)

The C++ version of ops\_dat\_set\_data\_memspace where the arguments the same except no need of the ops\_dat arguments.

## 4.3.9 Linear algebra solvers

### Tridiagonal solver

This section specifies APIs that allow [Tridsolver](#) (a tridiagonal solver library) to be called from OPS. The library can be used to solve a large number of tridiagonal systems of equations stored in multidimensional datasets. Parameters that are passed to Tridsolver from OPS are stored in an ops\_tridsolver\_params object. The constructor for this class takes the ops\_block that the datasets are defined over as an argument and optionally also a solving strategy to use (only relevant to MPI applications). The following solving strategies are available (see Tridsolver for more details about these):

- GATHER\_SCATTER (not available for GPUs)
- ALLGATHER
- LATENCY\_HIDING\_TWO\_STEP
- LATENCY\_HIDING\_INTERLEAVED
- JACOBI
- PCR (default)

Then parameters specific to different solving strategies can be set using setter methods. For applications using MPI, it is beneficial to reuse ops\_tridsolver\_params objects between solves as much as possible due to set up times involved with creating Tridsolver's MPI communicators.

### ops\_tridMultiDimBatch

**void ops\_tridMultiDimBatch(int ndim, int solvedim, int\* range, ops\_dat a, ops\_dat b, ops\_dat c, ops\_dat d, ops\_tridsolver\_params \*tridsolver\_ctx)**

This solves multiple tridiagonal systems of equations in multidimensional datasets along the specified dimension. The matrix is stored in the a (bottom diagonal), b (central diagonal) and c (top diagonal) datasets. The right hand side is stored in the d dataset and the result is also written to this dataset.

| Arguments       | Description   |
|-----------------|---|
| ndim            | the dimension of the datasets   |
| solvedim        | the dimension to solve along  |
| range           | the range to solve over, similar to ops_par_loop's range argument, cannot exclude an entire MPI process |
| a               | the dataset for the lower diagonal  |
| b               | the dataset for the central diagonal  |
| c               | the dataset for the upper diagonal  |
| d               | the dataset for the right hand side, also where the solution is written to                              |
| trid-solver_ctx | an object containing the parameters for the Tridsolver library  |

### ops\_tridMultiDimBatch\_Inc

**void ops\_tridMultiDimBatch(int ndim, int solvedim, int\* range, ops\_dat a, ops\_dat b, ops\_dat c, ops\_dat d, ops\_dat u, ops\_tridsolver\_params \*tridsolver\_ctx)**

This solves multiple tridiagonal systems of equations in multidimensional datasets along the specified dimension. The matrix is stored in the a (bottom diagonal), b (central diagonal) and c (top diagonal) datasets. The right hand side is stored in the d dataset and the result is added to the u dataset.

| Arguments       | Description   |
|-----------------|---|
| ndim            | the dimension of the datasets   |
| solvedim        | the dimension to solve along  |
| dims            | the range to solve over, similar to ops_par_loop's range argument, cannot exclude an entire MPI process |
| a               | the dataset for the lower diagonal  |
| b               | the dataset for the central diagonal  |
| c               | the dataset for the upper diagonal  |
| d               | the dataset for the right hand side   |
| u               | the dataset that the solution is added to   |
| trid-solver_ctx | an object containing the parameters for the Tridsolver library  |

## 4.4 Runtime Flags and Options

The following is a list of all the runtime flags and options that can be used when executing OPS generated applications.

- `OPS_DIAGS=` : set OPS diagnostics level at runtime.
  - `OPS_DIAGS=1` - no diagnostics, default level to achieve the best runtime performance.
  - `OPS_DIAGS>1` - print block decomposition and `ops_par_loop` timing breakdown.
  - `OPS_DIAGS>4` - print intra-block halo buffer allocation feedback (for OPS internal development only).
  - `OPS_DIAGS>5` - check if intra-block halo MPI sends depth match MPI receives depth (for OPS internal development only).
- `OPS_BLOCK_SIZE_X=`, `OPS_BLOCK_SIZE_Y=` and `OPS_BLOCK_SIZE_Z=` : The CUDA (and OpenCL) thread block sizes in X, Y and Z dimensions. The sizes should be an integer between 1 - 1024, and currently they should be selected such that `OPS_BLOCK_SIZE_XOPS_BLOCK_SIZE_YOPS_BLOCK_SIZE_Z < 1024`
- `-gpudirect` : Enable GPU direct support when executing MPI+CUDA executables.
- `OPS_CL_DEVICE=` : Select the OpenCL device for execution. Usually `OPS_CL_DEVICE=0` selects the CPU and `OPS_CL_DEVICE=1` selects GPUs. The selected device will be reported by OPS during execution.
- `OPS_TILING` : Execute OpenMP code with cache blocking tiling. See the [Performance Tuning](#) section.
- `OPS_TILING_MAXDEPTH=` : Execute MPI+OpenMP code with cache blocking tiling and further communication avoidance. See the [Performance Tuning](#) section.

## 4.5 Doxygen

Doxygen generated from OPS source can be found [here](#).





## **EXAMPLES**

See `OPS/apps/[c|fortran]/[application]/test.sh` on compiling and running various parallel versions generated by OPS for each application. See the [OPS-APPS](#) repository to see the latest generated parallel code for each application.

Further documentation under construction.



## PERFORMANCE TUNING

### 6.1 Executing with GPUDirect

GPU direct support for MPI+CUDA, to enable (on the OPS side) add **-gpudirect** when running the executable. You may also have to use certain environmental flags when using different MPI distributions. For an example of the required flags and environmental settings on the Cambridge Wilkes2 GPU cluster see: <https://docs.hpc.cam.ac.uk/hpc/user-guide/performance-tips.html>

### 6.2 Cache-blocking Tiling

OPS has a code generation (`ops_gen_mpi_lazy`) and build target for tiling. Once compiled, to enable, use the `OPS_TILING` runtime parameter. This will look at the L3 cache size of your CPU and guess the correct tile size. If you want to alter the amount of cache to be used for the guess, use the `OPS_CACHE_SIZE=XX` runtime parameter, where the value is in Megabytes. To manually specify the tile sizes, use the `OPS_TILESIZE_X`, `OPS_TILESIZE_Y`, and `OPS_TILESIZE_Z` runtime arguments.

When MPI is combined with OpenMP tiling can be extended to the MPI halos. Set `OPS_TILING_MAXDEPTH` to increase the halo depths so that halos for multiple `ops_par_loops` can be exchanged with a single MPI message (see [TPDS2017](#) for more details) To test, compile CloverLeaf under `OPS/apps/c/CloverLeaf`, modify `clover.in` to use a  $6144^2$  mesh, then run as follows: For OpenMP with tiling:

```
export OMP_NUM_THREADS=xx; numactl -physnodebind=0 ./cloverleaf_tiled OPS_TILING
```

For MPI+OpenMP with tiling:

```
export OMP_NUM_THREADS=xx; mpirun -np xx ./cloverleaf_mpi_tiled OPS_TILING OPS_TILING_
↪MAXDEPTH=6
```

To manually specify the tile sizes (in number of grid points), use the `OPS_TILESIZE_X`, `OPS_TILESIZE_Y`, and `OPS_TILESIZE_Z` runtime arguments:

```
export OMP_NUM_THREADS=xx; numactl -physnodebind=0 ./cloverleaf_tiled OPS_TILING OPS_
↪TILESIZE_X=600 OPS_TILESIZE_Y=200
```

## 6.3 OpenMP and OpenMP+MPI

It is recommended that you assign one MPI rank per NUMA region when executing MPI+OpenMP parallel code. Usually for a multi-CPU system a single CPU socket is a single NUMA region. Thus, for a 4 socket system, OPS's MPI+OpenMP code should be executed with 4 MPI processes with each MPI process having multiple OpenMP threads (typically specified by the `OMP_NUM_THREAD` flag). Additionally on some systems using `numactl` to bind threads to cores could give performance improvements (see `OPS/scripts/numawrap` for an example script that wraps the `numactl` command to be used with common MPI distributions).

## 6.4 CUDA arguments

The CUDA (and OpenCL) thread block sizes can be controlled by setting the `OPS_BLOCK_SIZE_X`, `OPS_BLOCK_SIZE_Y` and `OPS_BLOCK_SIZE_Z` runtime arguments. For example,

```
./cloverleaf_cuda OPS_BLOCK_SIZE_X=64 OPS_BLOCK_SIZE_Y=4
```

## 6.5 OpenCL arguments

`OPS_CL_DEVICE=XX` runtime flag sets the OpenCL device to execute the code on.

Usually `OPS_CL_DEVICE=0` selects the CPU and `OPS_CL_DEVICE=1` selects GPUs.

## DEVELOPER GUIDE

Under construction.

### 7.1 Contributing

To contribute to OPS please use the following steps :

1. Clone the [OPS](#) repository (on your local system).
2. Create a new branch in your cloned repository
3. Make changes / contributions in your new branch
4. Submit your changes by creating a Pull Request to the `develop` branch of the OPS repository

The contributions in the `develop` branch will be merged into the `master` branch as we create a new release.



## PUBLICATIONS

See [OP-DSL publications page](#).





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`