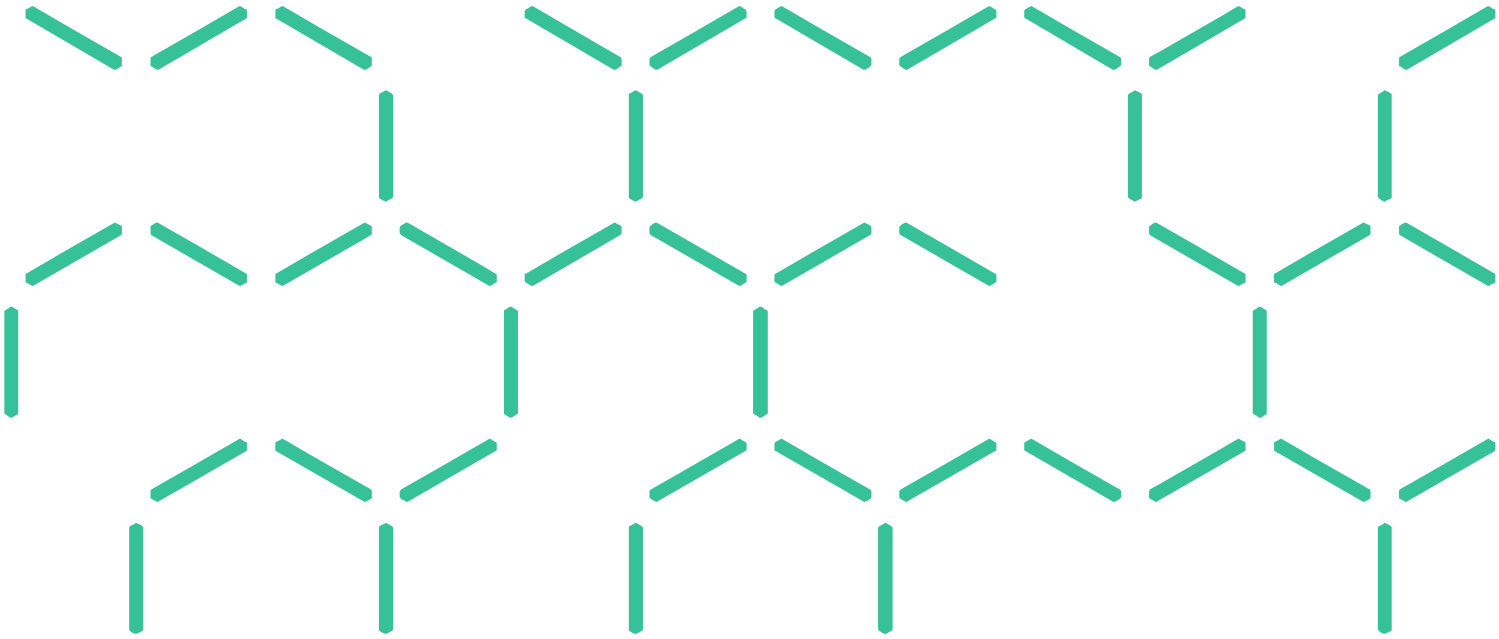
Spark processor programming guide

Guide to creating custom processor plug-ins for Spark

James Hilton and William Swedosh

EP17523
January 2017

Spark version 0.9.3



Citation

Hilton JE and Swedosh W (2017) Spark processor programming guide 0.9.3. CSIRO, Australia.

Copyright

Important disclaimer

Contents

Figures

1 Summary

Spark is a toolkit for simulating the spread of wildfires over terrain. The toolkit consists of a number of modules specifically designed for wildfire spread. These include readers and writers for geospatial data, a computational model to simulate a propagating front, a range of visualisations and tools for analysing the resulting data.

This document provides a guide to creating custom user-defined processors (Spark sub-solvers) using C++, Workspace and the spark-tools package.

OpenCL

Spark requires OpenCL 1.2 to run. This now comes as standard with all Windows and Mac graphics drivers. Please update your graphics driver before installing Spark to ensure that the latest OpenCL version is installed.

2 Introduction

Spark is a configurable system for predicting the evolution of a fire perimeter over a landscape based on empirical rate-of-spread models. A key aspect of Spark is configurability. Spark has been designed to handle multiple rate-of-spread models for different fuel types. Spark has also been designed to be compatible with future fire models and new types of fire behaviour.

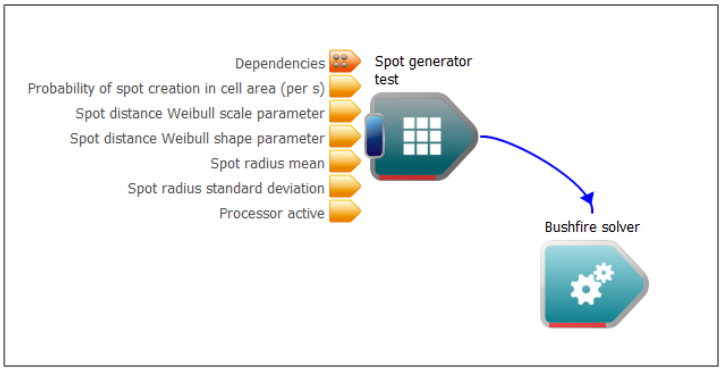


Figure 1 - Processor plug-in for Spark solver

The configurability of Spark is implemented using a modular approach. A scenario, such as a fire event, is broken into discrete processing blocks which can be chained together. For example, input data layers may be read by one block, the prediction of the fire perimeter calculated by a subsequent block and the predicted fire extent written or analysed by a further set of processing blocks.

These processing blocks can be chained together in a visual environment called the CSIRO Workspace[1]. This visual environment also includes a range of visualisation and analysis tools for interrogating and visualising data sets, making the system a powerful and easy-to-use environment for rapid prototyping, research and data analysis. Spark and Workspace also contain a comprehensive library of GIS processing and visualisation tools suitable for natural hazards.

A key aspect of Workspace and Spark is that the tools maintain, wherever possible, an open development environment based on open-source software. This allows users to write custom plug-ins and code for Workspace and Spark. These modules must be written in C++ and use the Workspace Qt-based environment.

This document provides a detailed guide to writing a Spark sub-solver or *processor*. A sub-solver is executed at every simulation time step and has free access to all of the internal data layers and variables used within the simulation. Processors provide fine grained control of the simulation and can be used to provide dynamic fire behaviour effects that are not modelled by the core rate-of-spread simulation. Examples include spot-fire generation, wind field generation and application to the fire perimeter, pressure feedback effects from the fire and disruption effects.

Processors provide an extra level of low-level configurability to the simulation. However, they are intended for expert users familiar with numerical modelling as incorrect use could destabilise the simulation or even the computational instance in which they are run.

[1] Information and the latest version of Workspace can be found on the Workspace research site: https://research.csiro.au/workspace/

3 Prerequisites

Workspace plug-ins, including Spark processors, are written in C++ and require at least general knowledge of C. The system uses the Qt environment which adds a large amount of additional functionality to standard C++. Knowledge of Qt is not necessary for programming processors, although may be desirable as many common data structures and libraries are implemented in Qt. Underlying projects are built using CMake, a freely available cross-platform build manager.

Computation within Spark uses OpenCL, a cross platform parallel processing environment. Processors do not have to use OpenCL for their own computation but require an OpenCL environment to be present on the system. Visualisation is carried out using OpenGL based on an internal Workspace format. Knowledge of these are not required unless integration with visualisation is required for the processor.

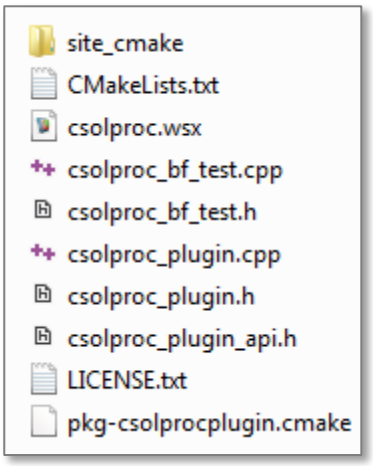
The software requirements are:

1. A C++ development environment with C++11 support. This guide is based on Windows Visual Studio Express 2013, which is freely available from Microsoft.
2. An installation of Workspace 4 or above. This is freely available from the CSIRO Workspace site.
3. An installation of spark-tools 0.9.3 or above. This is free for scientific researchers and research agencies, please contact us for details on obtaining a copy.
4. A development environment for OpenCL. Spark uses OpenCL for GPU computation and it is necessary to link to the OpenCL libraries. There are many freely available packages depending on your computer set-up.
    - AMD OpenCL SDK, works on all platforms and supports AMD graphics cards.
    - Intel OpenCL SDK, for Intel CPUs and graphics cards.
    - NVIDIA CUDA toolkit, for NVIDIA graphics cards.
5. The latest version of CMake, available from the development site.
6. (Optional) example code for this draft, available from the Spark site. The code is available as the ‘Spark processor development example’.

4 Building the example

These steps assume that you're using Visual Studio 2013 express, have downloaded the example code and installed Workspace, spark-tools and CMake.

1. Unzip the example code to a suitable location. The directory contents should be as follows:



   These files are:

   a. `site_cmake`: A directory containing CMake scripts specific to Workspace. This usually will not need to be edited.

   b. `CMakeLists.txt`: A script for CMake to build the project.

   c. `csolproc.wsx`: A test Workspace file for the processor.

   d. `csolproc_bf_test.cpp`: The C++ code for the processor.

   e. `csolproc_bf_test.h`: The C++ header file for the processor.

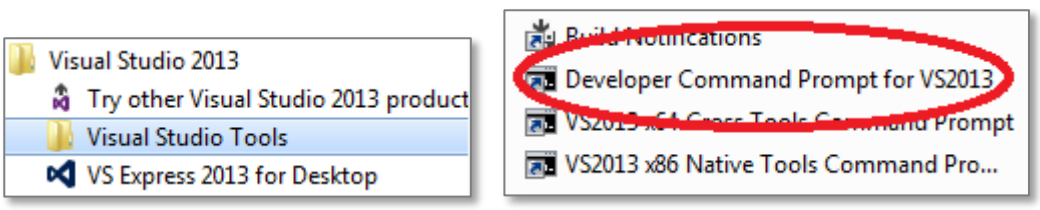
   f. `csolproc_plugn.cpp`: The C++ code for integration in Workspace.

   g. `csolproc_plugin.h`: The C++ header for integration in Workspace.

   h. `LICENSE.txt`: The code license. Please note this is a variant of the MIT/BSD license, please read this file for full details.

   i. `pkg-csolprocplugin.cmake`: A project-specific CMake build script.

2. Run CMake. This must be run from within a Visual Studio command prompt to set specific development environment variables. The tools are available on Windows under the start menu, under the 'Visual Studio 2013' folder.



In the command prompt, CMake must be launched from Workspace. Again, this is ensure certain Workspace-specific development variables are set. CMake can be launched using the command:

```
"C:\Program Files\csiro.au\workspace\bin\workspace-batch.exe" --launch cmake-gui
```

If Workspace is installed under a different location the command should be modified as necessary. CMake gui should open if successful:

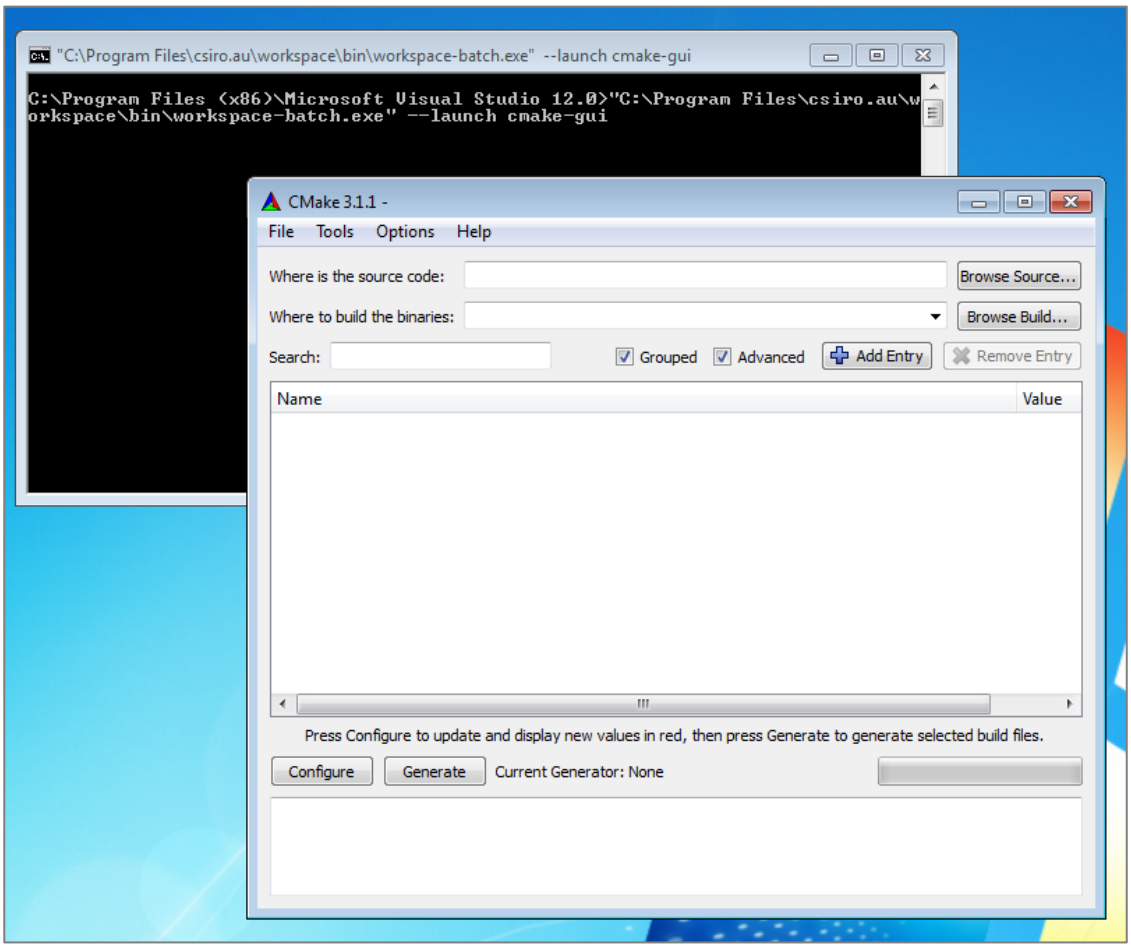


Figure 2 - Visual Studio Command Prompt and CMake

3. Configure CMake. First, select the directory where the source code was unzipped under the 'Where is the source code' field. Next, choose a location to build the project. This can be in any location, but has been chosen to be in the same directory as the code in a /build directory. Once these have been chosen, press the 'Configure' button. A prompt will appear if the build directory needs to be created. Click 'Yes' to create the directory.

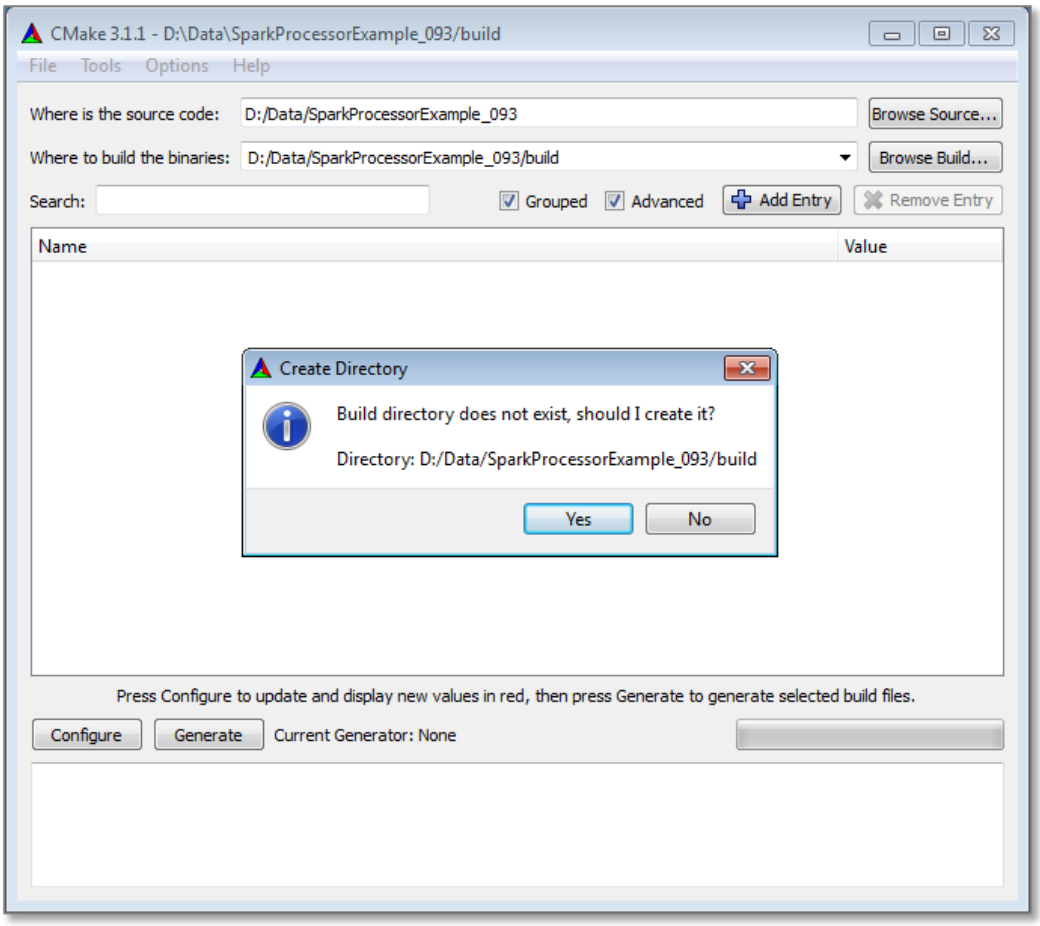


Figure 3 - CMake directory set-up and creation prompt.

CMake will then ask for the compiler to use. Select 'Visual Studio 12 2013 Win64', 'Use default native compilers' and 'Finish'. CMake will take a few moments to build.

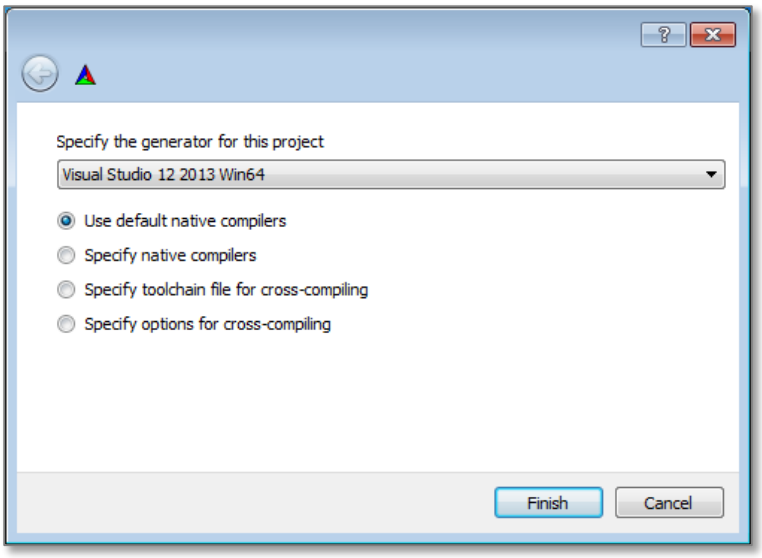


Figure 4 - Compiler configuration in CMake.

4. Resolving paths. CMake will give with warnings and show several red rows. The red rows indicate new variables that have been created and the error is because some link paths need to be manually entered. Click OK to dismiss the error.

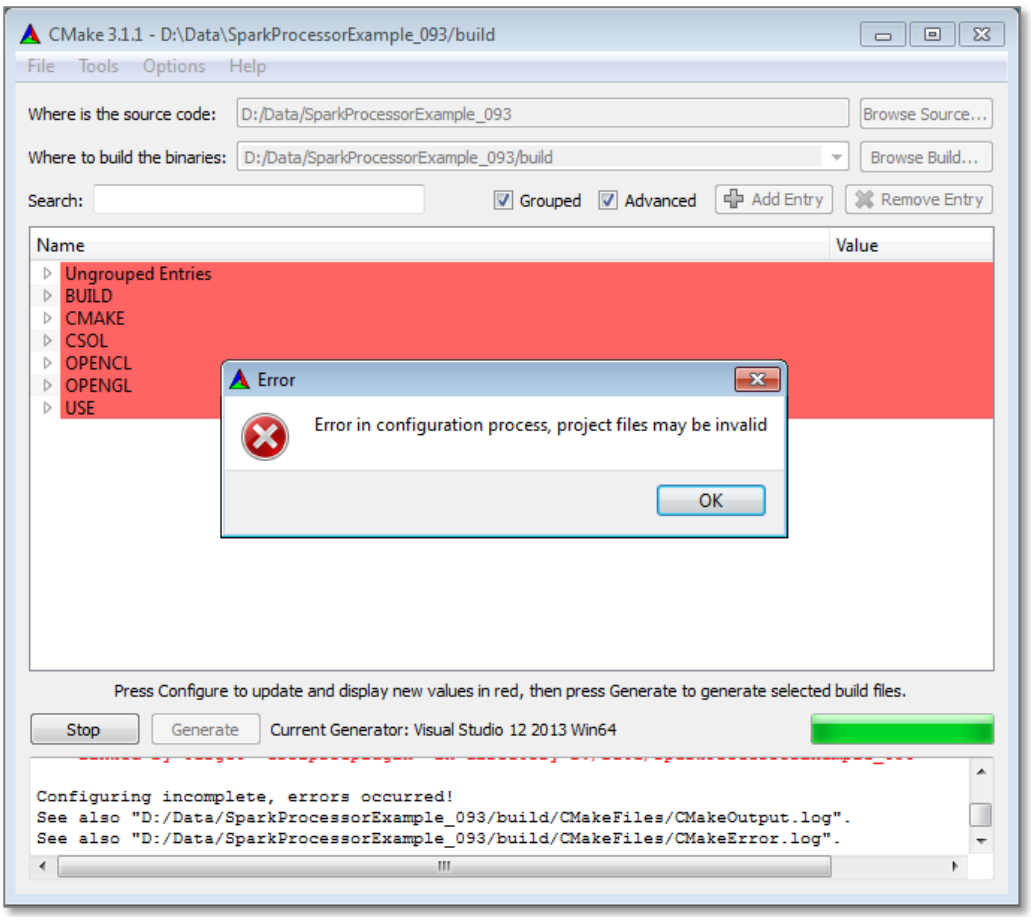


Figure 5 - CMake creation error due to missing links.

There are two main library paths which may cause this error, the spark-tools library path and the OpenCL library path. Expand the CSOL and OPENCL rows in CMake (using the small triangular arrows at the start of each row).

a. Change `CSOL_INCLUDE_DIR-NOTFOUND` to the directory containing the spark tools include headers. These will be in the spark-tools installation folder, for example `C:/Program Files/csiro.au/Spark_tools_fsp-vs2013/include`

b. Change `CSOL_LIBRARIES-NOTFOUND` to the spark-tools main library, csolplugin. This will be in the spark-tools installation folder, for example `C:/Program Files/csiro.au/Spark_tools_fsp-vs2013/lib/Plugins/csolplugin.dll`

c. Change `OPENCL_INCLUDE_DIRS-NOTFOUND` to the location of the OpenCL headers within your OpenCL development. For example the NVIDIA SDK headers are located at: `C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v7.5/include`

d. Change `OPENCL_LIBRARIES-NOTFOUND` to the location of the OpenCL library. For example the NVIDIA SDK library file is located at:

`C:/Program Files/NVIDIA GPU Computing Toolkit/CUDA/v7.5/lib/x64/OpenCL.lib`

Finally, you must change the `OPENCL_FLOAT_TYPE` to **float** if you are using the standard spark-tools package.

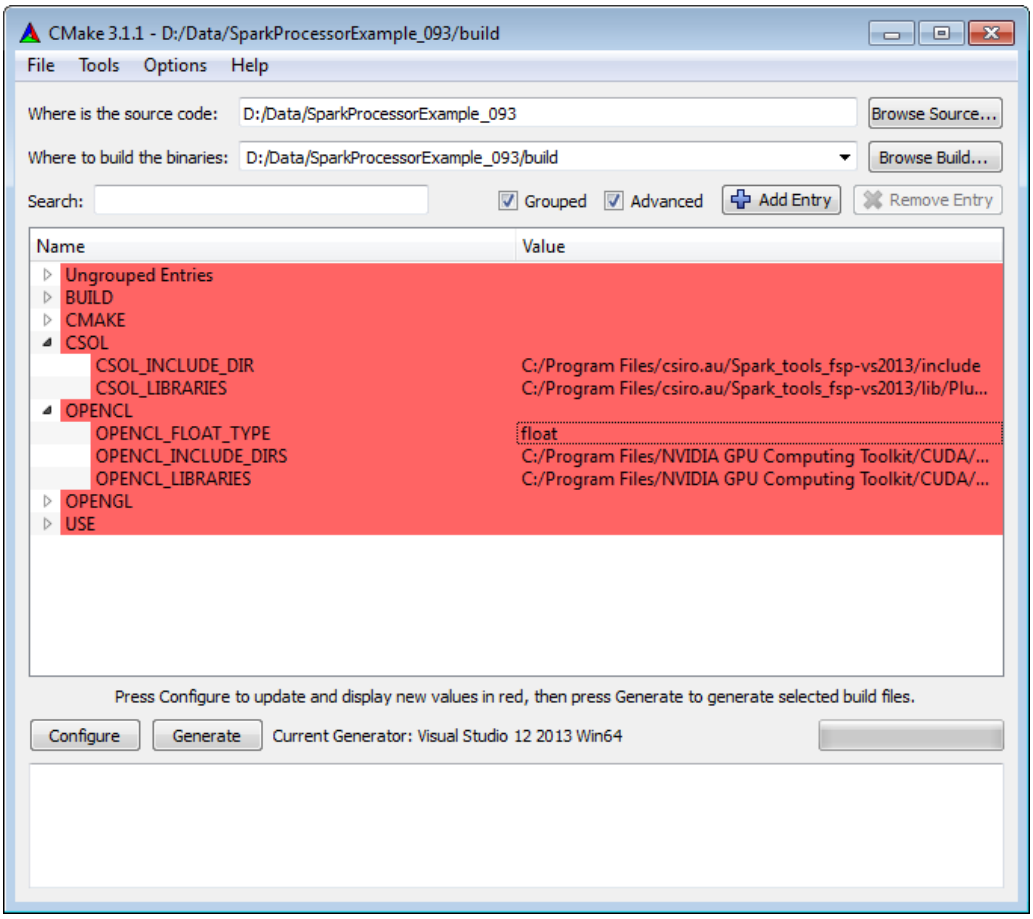


Figure 6 - CMake configuration with CSol (Spark) and OpenCL paths set.

Once the directories are set press 'Generate' to generate the project. If the project is created successfully the log will end with 'Configuring done', followed by 'Generating done'.

OpenCL floating point type

Spark uses 32-bit floating point by default for maximum compatibility across compute units. The OPENCL_FLOAT_TYPE must be set to **float** to avoid linker errors.

5. Open the project. To build a Workspace project Visual Studio must be launched by Workspace. This is to ensure various development options are set up for the build. Visual studio can be launched using, for example:

```
"C:\Program Files\csiro.au\workspace\bin\workspace-batch.exe" --launch
"C:\Program Files (x86)\Microsoft Visual Studio
12.0\Common7\IDE\WDExpress.exe"
```

Once launched, go to 'FILE/Open Project' and navigate to the build directory in which you created your project. This is the directory specified in the 'Where to build the binaries' field in CMake. Then open '`CSOLPROC.sln`', the Visual Studio project file.

The source files are available as a tree in the 'Solution Explorer' in the right hand dock. To build the plugin, ensure the Solution Configuration is 'Release' rather than 'Debug' from the drop-down list in the toolbar. Next for to 'BUILD" and select 'Rebuild Solution' (or press F7). If the plugin is built successfully, the log should show:

```
========== Build: 2 succeeded, 0 failed, 0 up-to-date, 2 skipped ==========
```

6. Finally, to connect the plugin to Workspace, start Workspace and go to 'Settings', 'Configure application'. Choose 'Plugins' from the list on the left-hand side and pick 'Add' to the right of 'Plugin libraries and search paths. Select the library file for the processor under your build directory. In this example the library is located at:

```
D:/Data/SparkProcessorExample_093/build/Install/lib/Plugins/
csolprocplugin.dll
```

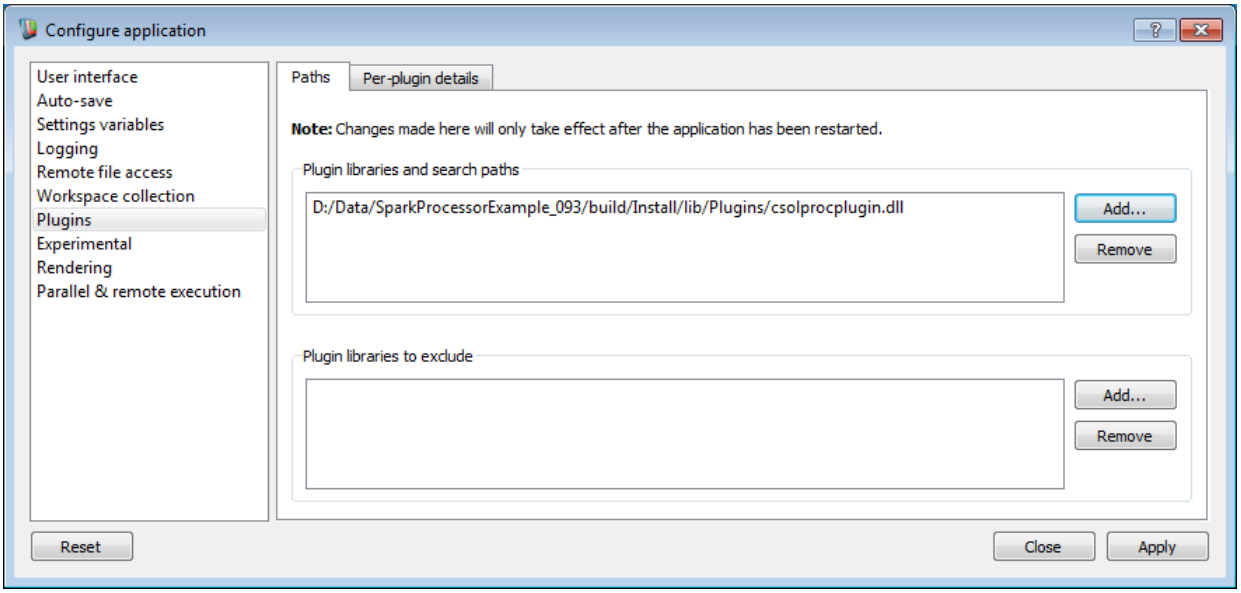


Figure 7 - Workspace configuration

Click 'Apply' to apply these settings and 'Close' to close the configuration window. Then re-start workspace for the plugin to load.

If everything has worked correctly, the log should contain:

```
Adding CSolProc version 0.1.0
```

A BushfireSpottingProcessorTest operation should also appear in the Workspace catalogue. To test the plugin, load and run the csolproc.wsx file. This should give a similar output to the following:

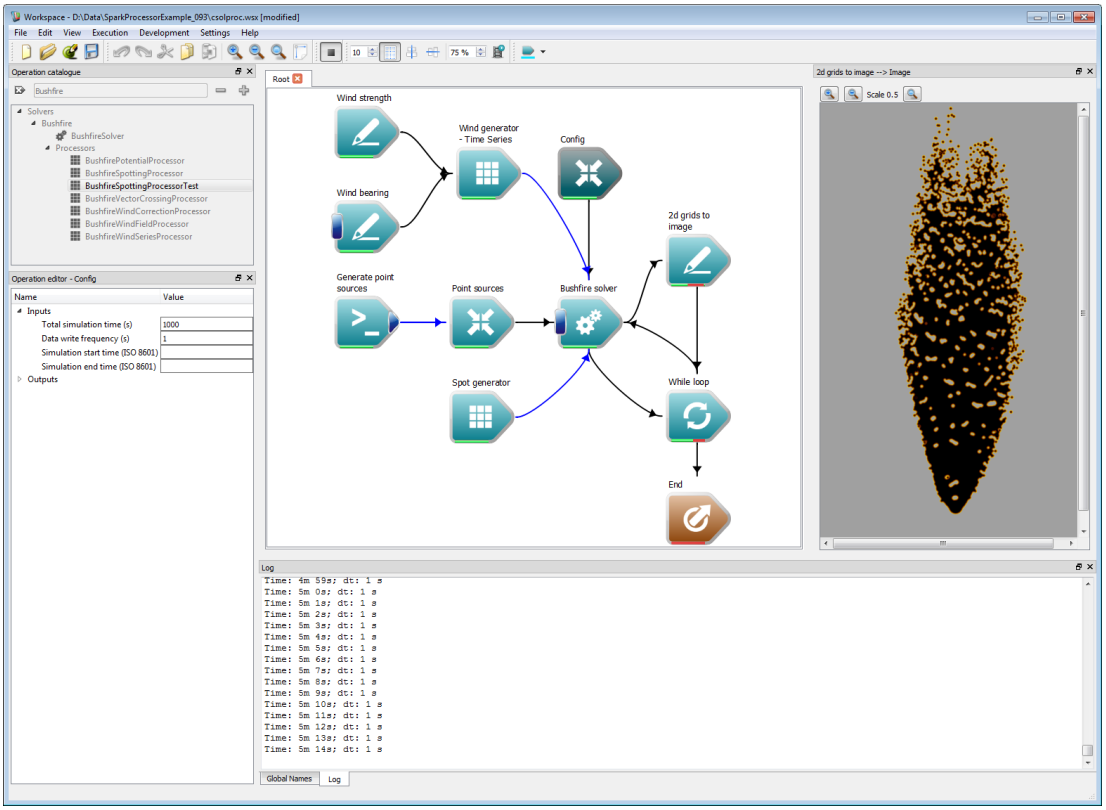


Figure 8 - Spot processor test workflow.

Visual Studio

Once these steps are carried out they do not need to be repeated and the project can simply be opened in Visual Studio. However, Visual Studio **must** be launched from Workspace each time using the command given in step 5. This ensures linker paths are set correctly.

5 Processor programming

Processors are sub-solvers scheduled and run from the main Spark solver that have complete control over the current fields and variables within the solver. They are intended as modular processing blocks capable of implementing advanced computational models beyond the scope of Workspace operations.

Processors have two main functions:

```
init(CSol::CSol_Solver &solver)
```

which is called to initialise the processor and overloaded

```
operator(CSol::CSol_Solver &solver, std::map<long long, CSol::BF_Tile *> &tiles)
```

which is called at each time step. Both functions pass the current solver handle, `solver`, as the first argument. The required time-step can be polled from this solver handle; processors are free to implement as many sub-steps or operations as required as long as their internal time state matches the solver time state at the end of execution.

The execution function additionally passes a `std::map` of the tiles in the solver, `tiles`. The map key is a unique ID for the tile and the map value is a pointer to the tile. Usually all tiles should be iterated over to process the entire fire domain. Tiles that are completely burnt are flagged and may not need to be used. The processor can check is a tile is completely burnt using the `isAlive()` function on a tile. If this is `false`, the tile is burnt.

A typical processor will:

1. Get simulation variables from the solver.
2. Iterate over the file tiles, running a sub-solver on each tile.
3. Modify a solver buffer or variable.
4. Return control to the solver after processing all tiles.

One final function that must be implemented for processors is the `getPosition()` operation, which returns where the processor should be executed within the solver time step. The choices are given in Table 1.

NAME	DESCRIPTION
ProcPosition::None	Processor is skipped.
ProcPosition:: Before_Time_Integration	Processor is run at the start of the time step.
ProcPosition:: After_Time_Integration	Processor is run at the end of the time step.

Table 1 - Processor operation positions

Information about the simulation variables can be requested from the solver instance passed in the initialisation and operation functions. A summary of the solver functions and information types that can requested are given in Table 2.

NAME	DESCRIPTION
getMetrics(hx, hy, ox, oy)	The layer cell size, hx and hy, and the global offset of the layer, ox and oy.
getDimensions(xTiles, yTiles, tileSize, localSize)	The number of fire tiles in the x (xTiles) and y (yTiles) directions, the number of cells in each (square) tile and the recommended size of an OpenCL work-group (localSize).
getVar(CSol::CSol_Solver::Var_Type_TimeStep, dt)	The current time step, dt.
getVar(CSol::CSol_Solver:: Var_Type_Time, time)	The current solver time in seconds since simulation start.
getVar(CSol::CSol_Solver:: Var_Type_Seed, seed)	The current random seed for the simulation.
std::vector<double> &getVect(CSol_Solver::Vect_Type_x) **std::vector<double> &getVect(CSol_Solver::Vect_Type_y)** **std::vector<double> &getVect(CSol_Solver::Vect_Type_r)** **std::vector<double> &getVect(CSol_Solver::Vect_Type_t)**	Four std::vector<double> variables representing the x and y positions of spot fire sources started at time t with radius r.

Table 2 - Solver information functions

The layer data for each tile can be read directly from the tiles using the functions given in Table 3.

NAME	DESCRIPTION
cl::Buffer *const get_pbf0()	Returns tile front buffer
cl::Buffer *const get_pba()	Returns arrival time buffer
cl::Buffer *const get_pbs()	Returns speed buffer
cl::Buffer *const get_pbrand()	Returns tile random buffer
cl::Buffer *const get_pbc()	Returns tile classification buffer
cl::Buffer *const get_pbpoi()	Returns points-of-interest raster buffer
cl::Buffer *const get_pbw_u()	Returns tile wind u (x-component) buffer
cl::Buffer *const get_pbw_v()	Returns tile wind v (y-component) buffer
cl::Buffer *const get_pbdata()	Returns tile data buffer
cl::Buffer *const get_pbconst()	Returns tile constant buffer
cl::Buffer *const get_pbx()	Returns user-defined buffers

Table 3 - Tile data buffer functions

Further information on the CSol data types can be found in the Spark toolkit user guide.

Memory access

Layer data from tiles are passed as direct pointers for maximum efficiency. Care must be taken not to deallocate or re-map this pointer.

CONTACT US

t 1300 363 400
+61 3 9545 2176
e csiroenquiries@csiro.au
w www.data61.csiro.au

AT CSIRO WE SHAPE THE FUTURE

We do this by using science and technology to solve real issues. Our research makes a difference to industry, people and the planet.

FOR FURTHER INFORMATION

James Hilton
Senior Research Scientist
t +61 3 9518 5974
e james.hilton@csiro.au