UFO Documentation

Release 0.13.0

Matthias Vogelgesang

Aug 21, 2017

Contents

CHAPTER 1

Users

Installation

In this section, information about how to install the UFO library and the accompanying filter suite is described.

Installation on Linux

Installing RPM packages

We provide RPM-based packages for openSUSE, RHEL and CentOS7. First you have to add the openSUSE repository matching your installation. Go to our OBS page, copy the URL that says "Go to download repository" and use that URL with:

```
opensuse $ zypper addrepo <URL> repo-ufo-kit
centos $ wget <URL> -O /etc/yum.repos.d/ufo-kit.repo
```

Now update the repositories and install the framework and plugins:

```
opensuse $ zypper install ufo-core ufo-filters
centos $ yum install ufo-core ufo-filters
```

Installing from source

If you want to build the most recent development version, you have to clone the source from our repository, install all required dependencies and compile the source.

Retrieving the source code

In an empty directory, issue the following commands to retrieve the current unstable version of the source:

```
$ git clone https://github.com/ufo-kit/ufo-core
$ git clone https://github.com/ufo-kit/ufo-filters
```

The latter is used for developers who have write-access to the corresponding repositories. All stable versions are tagged. To see a list of all releases issue:

```
$ git tag -l
```

Installing dependencies

UFO has only a few hard source dependencies: GLib 2.0, JSON-GLib 1.0 and a valid OpenCL installation. Furthermore, it is necessary to build the framework with a recent version of CMake. Sphinx is used to create this documentation. This gives you a bare minimum with reduced functionality. To build all plugins, you also have to install dependencies required by the plugins.

OpenCL development files must be installed in order to build UFO. However, we cannot give general advices as installation procedures vary between different vendors. However, our CMake build facility is in most cases intelligent enough to find header files and libraries for NVIDIA CUDA and AMD APP SDKs.

Ubuntu/Debian

On Debian or Debian-based system the following packages are required to build ufo-core:

```
$ sudo apt-get install build-essentials cmake libglib2.0-dev libjson-glib-dev
```

You will also need an OpenCL ICD loader. To simply get the build running, you can install

```
$ sudo apt-get install ocl-icd-opencl-dev
```

Generating the introspection files for interfacing with third-party languages such as Python you must install

```
$ sudo apt-get install gobject-introspection libgirepository1.0-dev
```

and advised to install

```
$ sudo apt-get install python-dev
```

To use the `ufo-mkfilter` script you also need the jinja2 Python package:

```
$ sudo apt-get install python-jinja2
```

Building the reference documentation and the Sphinx manual requires:

```
$ sudo apt-get install gtk-doc-tools python-sphinx
```

Additionally the following packages are recommended for some of the plugins:

```
$ sudo apt-get install libtiff5-dev
```

openSUSE and CentOS7

For openSUSE (zypper) and CentOS7 the following packages should get you started:

```
$ [zypper|yum] install cmake gcc gcc-c++ glib2-devel json-glib-devel
```

Additionally the following packages are recommended for some of the plugins:

```
$ [zypper|yum] install libtiff-devel
```

Building ufo-core

Change into another empty *build* directory and issue the following commands to configure

```
$ cmake <path-to-ufo>
```

CMake will notify you, if some of the dependencies are not met. In case you want to install the library system-wide on a 64-bit machine you should generate the Makefiles with

```
$ cmake -DCMAKE_INSTALL_LIBDIR=lib64 <path-to-ufo>
```

For earlier versions of PyGObject, it is necessary that the introspection files are located under `/usr` not `/usr/local`. You can force the prefix by calling

```
$ cmake -DCMAKE_INSTALL_PREFIX=/usr <path-to-ufo>
```

Last but not least build the framework, introspection files, API reference and the documentation using

```
$ make
```

You should now run some basic tests with

```
$ make test
```

If everything went well, you can install the library with

```
$ make install
```

See also:

Why can't the linker find libufo.so?

Building ufo-filters

Once ufo-core is installed you can build the filter suite in a pretty similar way

```
$ mkdir -p build/ufo-filters
$ cd build/ufo-filters
$ cmake <path-to-ufo-filters> -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_INSTALL_LIBDIR=lib64
$ make
$ make install
```

Python support

ufo-core has GObject introspection to let third-party languages interface with the library. To build the support files you need the GObject introspection scanner `g-ir-scanner` and compiler `g-ir-compiler` which you can get on openSUSE via

```
$ zypper install gobject-introspection-devel python-gobject2
```

In the `python/` subdirectory of the source distribution, additional Python modules to interface more easily with the framework is provided. To install the NumPy module and the high-level interface run

```
$ cd python/ && python setup install
```

Refer to the README for additional information.

Installation on MacOS X

Preface: This information is kindly provided by Andrey Shkarin and Roman Shkarin.

1. Install the MacPorts from http://macports.org

   Note: If you previously installed MacPorts, and it can not be started after latest installation. *Error: port dlopen (...* You must download the tar.gz file and install it using a terminal:

   ```
   ./configure
   make
   sudo make install
   ```

   ---

2. Install the necessary packages through macports:

   ```
   sudo port install glib2
   sudo port install gtk2
   sudo port install json-glib
   ```

3. Install CMake from http://cmake.org

4. Make ufo-core

   (a) Got to the directory ufo-core and run:

   ```
   export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
   cmake .
   ```

   (b) Run:

   ```
   make
   sudo make install
   ```

   (c) Installation is complete, perhaps the last lines are as follows:

   ```
   -- Installing: /usr/local/lib/pkgconfig/ufo.pc
   CMake Error at src/bindings/cmake_install.cmake:33 (FILE):
   file INSTALL cannot find
     "/Users/Andrey/Desktop/ufo-distr/ufo-core/src/bindings/../src/Ufo-0.1.gir".
   Call Stack (most recent call first):
     src/cmake_install.cmake:60 (INCLUDE)
     cmake_install.cmake:32 (INCLUDE)
   make: *** [install] Error 1
   ```

5. Make filters

1. Go to ufo-filters directory. Now, since libufo was installed in lib64, we must update the paths to look for shared libraries:

```
export DYLD_LIBRARY_PATH=/usr/local/lib:/usr/local/lib64
export PKG_CONFIG_PATH=/usr/local/lib/pkgconfig
```

   (a) Run:

```
cmake .
make
sudo make install
```

2. Build the test project and verify that everything works.

Installation with Docker

Install Docker

See Docker documentation to install Docker on your system.

Build Dockerfile

Download AMD-APP-SDK-v3.0-0.113.50-Beta-linux64, paste it to the directory with Dockerfile in it and run:

```
$ docker build -t docker .
```

This will build the image of opensuse 13.1, install all dependencies, AMD drivers, ufo-core, ufo-filters and ufo-ir on it.

Use Docker

Run installed image in a new Docker container and use gpu in it:

```
$ docker run -i -t --device=/dev/ati/card0 docker /bin/bash
```

See all created containers:

```
$ docker ps
```

See all created images:

```
$ docker images
```

Stop a running container:

```
$ docker stop #ID_or_container_name
```

Start a stopped container:

```
$ docker start #ID_or_container_name
```

Attach to a running container:

```
$ docker attach #ID_or_container_name
```

For more help run:

```
$ docker --help
```

If you need the container IP, call in a running container:

```
$ cd etc
$ vi hosts
```

Using UFO

UFO is a framework for high-speed image processing at Synchrotron beamlines. It facilitates every available hardware device to process tomographic data as fast as possible with on-line reconstruction as the ultimate goal.

It is written in C using the GLib and GObject libraries to provide an object-oriented API.

After *installing* the framework you're ready to build your own image processing pipeline or implement a new filter.

Quick start guide

There are three ways to specify and execute a graph of tasks. The simplest method requires you to construct a pipeline on the command line using the `ufo-launch` tool which is similar to gst-launch from the GStreamer package. The second method involves writing a *JSON file* that is executed by the `ufo-runjson` utility, the other way uses the provided language bindings to setup the task graph specifically.

To influence the execution from the process environment check the existing *environment variables*.

Launching pipelines on the command line

The `ufo-launch` tool receives a list of tasks separated by exclamation marks `!` and executes the data flow in that order. To specify task parameters, you can add key-value pairs seperated by an equal sign. For example, to split a multi EDF file to single TIFFs you would do:

```
$ ufo-launch read path=file.edf ! write filename=out-%05i.tif
```

You can concatenate an arbitrary number of tasks. For example to blur the lena image you would something like this:

```
$ ufo-launch read path=lena.tif ! blur size=20 sigma=5 ! write
```

Some tasks receive multiple inputs which requires the use of brackets to collect all arguments. For example, a simple flat field correction would look like this:

```
$ ufo-launch [read path=radios, read path=darks, read path=flats]! flat-field-correct␣
↪! write filename=foo.tif
```

Using a JSON description

Our *UFO* JSON format has the advantage to be language-agnostic and portable across different versions of the UFO framework. Let's start with a simple example, that computes the one-dimensional Fourier-transform on a set of input files:

```
{
    "nodes" : [
        {
            "plugin": "reader",
            "name": "reader",
            "properties" : { "path": "*.tif" }
        },
        {
            "plugin": "fft",
            "name": "fft"
        }
        {
            "plugin": "writer",
            "name": "writer",
            "properties" : { "filename": "fft-%05i.tif" }
        }
    ],
    "edges" : [
        {
            "from": { "name": "reader" },
            "to": { "name": "fft" }
        },
        {
            "from": { "name": "fft" },
            "to": { "name": "writer" }
        }
    ]
}
```

Save this to a file named `fft.json` and execute it by calling the `ufo-runjson` tool:

```
$ ufo-runjson fft.json
```

C interface

A simple UFO program written in C that loads the JSON description can look like this:

```
/* ufo/ufo.h is the only header allowed to be included */
#include <ufo/ufo.h>

int main (void)
{
    UfoTaskGraph *graph;
    UfoScheduler *scheduler;
    UfoPluginManager *manager;

    g_type_init ();  /* you _must_ call this! */

    graph = UFO_TASK_GRAPH (ufo_task_graph_new ());
    manager = ufo_plugin_manager_new (NULL);

    ufo_task_graph_read_from_file (graph, manager, "hello-world.json", NULL);

    scheduler = ufo_scheduler_new (NULL, NULL);
    ufo_scheduler_run (scheduler, graph, NULL);
```

```
    /* Destroy all objects */
    g_object_unref (graph);
    g_object_unref (scheduler);
    g_object_unref (manager);
    return 0;
}
```

You can compile this with:

```
$ gcc `pkg-config --cflags --libs ufo` foo.c -o foo
```

As you can see we simply construct a new `UfoGraph` object from a JSON encoded *configuration file* and execute the computation pipeline with a `UfoScheduler` object.

Rather than loading the structure from a file, you can also construct it by hand:

```
#include <ufo/ufo.h>

int main (void)
{
    UfoTaskGraph *graph;
    UfoPluginManager *manager;
    UfoBaseScheduler *scheduler;
    UfoTaskNode *reader;
    UfoTaskNode *writer;

#if !(GLIB_CHECK_VERSION (2, 36, 0))
    g_type_init ();
#endif

    graph = UFO_TASK_GRAPH (ufo_task_graph_new ());
    manager = ufo_plugin_manager_new ();
    scheduler = ufo_scheduler_new ();
    reader = ufo_plugin_manager_get_task (manager, "read", NULL);
    writer = ufo_plugin_manager_get_task (manager, "write", NULL);

    g_object_set (G_OBJECT (reader),
                  "path", "/home/user/data/*.tif",
                  "number", 5,
                  NULL);

    ufo_task_graph_connect_nodes (graph, reader, writer);
    ufo_base_scheduler_run (scheduler, graph, NULL);
    return 0;
}
```

Python Interface

There are no plans to support any languages with manually written language bindings. However, UFO is a GObject-based library from which `gir` (GObject Introspection) files can be generated at build time. Any language that supports GObject Introspection and the `gir`/`typelib` format is thus able to integrate UFO. No manual intervention is need if the GObject Introspection tools are found.

Because several languages support GObject Introspection, you have to consult the appropriate reference manuals to find out how the GObjects are mapped to their language equivalents. Some of the options are

- Python: PyGObject

- Javascript: Gjs and Seed
- Vala has direct support using the `--pkg` option

A GNOME wiki page lists all available runtime bindings.

The simple example from the beginning – with Python-GObject installed – would look like this:

```
from gi.repository import Ufo

manager = Ufo.PluginManager()
graph = Ufo.TaskGraph()
scheduler = Ufo.Scheduler()

graph.read_from_json(manager, "some-graph.json")
scheduler.run(graph)
```

Similarly, constructing the graph by hand maps one-to-one to the Python object and keyword system:

```
from gi.repository import Ufo

graph = Ufo.Graph()
manager = Ufo.PluginManager()
scheduler = Ufo.Scheduler()

reader = manager.get_task('read')
writer = manager.get_task('write')
reader.set_properties(path='/home/user/data/*.tif', number=5)

graph.connect_nodes(reader, writer)
scheduler.run(graph)
```

Environment variables

You can modify the run-time behaviour by setting environment variables:

G_MESSAGES_DEBUG
: Controls the output of the library. By default nothing is printed on stdout. Set this to *all* to see debug output.

UFO_PLUGIN_PATH
: Colon-separated list of paths in which plugin manager looks for additional task modules. The plugins are load with descending priority.

UFO_DEVICES
: Controls which OpenCL devices should be used. It works similar to the *CUDA_VISIBLE_DEVICES* environment variables, i.e. set it to *0,2* to choose the first and third device that's available.

UFO_DEVICE_TYPE
: Controls which OpenCL device types should be considered for execution. The variable is a comma-separated list with strings being *cpu*, *gpu* and *acc*, i.e. to use both CPU and GPUs set *UFO_DEVICE_TYPE="cpu,gpu"*.

Task execution

This section provides a deeper look into the technical background concerning scheduling and task execution. The execution model of the UFO framework is based on the `Ufo.TaskGraph` that represents a network of interconnected task nodes and the `Ufo.BaseScheduler` that runs these tasks according to a pre-defined strategy. The `Ufo.Scheduler` is a concrete implementation and is the default choice because it is able to instantiate tasks in a

multi-GPU environment. For greater flexibility, the `Ufo.FixedScheduler` can be used to define arbitrary GPU mappings.

Profiling execution

By default, the scheduler measures the run-time from initial setup until processing of the last data item finished. You can get the time in seconds via the `time` property

```
g = Ufo.TaskGraph()
scheduler = Ufo.Scheduler()
scheduler.run(g)
print("Time spent: {}s".format(scheduler.time))
```

To get more fine-grained insight into the execution, you can enable tracing

```
scheduler.props.enable_tracing = True
scheduler.run(g)
```

and analyse the generated traces for OpenCL (saved in `opencl.PID.json`) and general events (saved in `trace.PID.json`). To visualize the trace events, you can either use the distributed `ufo-prof` tool or Google Chrome or Chromium by going to chrome://tracing and loading the JSON files.

Broadcasting results

Connecting a task output to multiple consumers will in most cases cause undefined results because some data is processed differently than others. A certain class of problems can be solved by inserting explicit `Ufo.CopyTask` nodes and executing the graph with a `Ufo.FixedScheduler`. In the following example, we want write the same data twice with a different prefix:

```
from gi.repository import Ufo

pm = Ufo.PluginManager()
sched = Ufo.FixedScheduler()
graph = Ufo.TaskGraph()
copy = Ufo.CopyTask()

data = pm.get_task('read')

write1 = pm.get_task('write')
write1.set_properties(filename='w1-%05i.tif')

write2 = pm.get_task('write')
write2.set_properties(filename='w2-%05i.tif')

graph.connect_nodes(data, copy)
graph.connect_nodes(copy, write1)
graph.connect_nodes(copy, write2)

sched.run(graph)
```

Note: The copy task node is not a regular plugin but part of the core API and thus cannot be used with tools like `ufo-runjson` or `ufo-launch`.

Running tasks in a cluster

The UFO framework comes with built-in cluster capabilities based on ZeroMQ 3.2. Contrary to bulk cluster approaches (e.g. solving large linear systems), UFO tries to distribute *streamed* data on a set of multiple machines. On each remote slave, `ufod` must be started. By default, the server binds to port 5555 on any available network adapter. To change this, use the `-l/--listen` option:

```
$ ufod --listen tcp://ib0:5555
```

will let `ufod` use the first Infiniband-over-IP connection.

On the master host, you pass the remote slave addresses to the scheduler object. In Python this would look like this:

```
sched = Ufo.Scheduler(remotes=['tcp://foo.bar.org:5555'])
```

Address are notated according to ZeroMQ.

Streaming vs. replication

Work can be executed in two ways: *streaming*, which means data is transferred from a master machine to all slaves and returned to the master after computation is finished and *replicated* in which each slaves works on its own subset of the initial input data. The former must be used if the length of the stream is unknown before execution, otherwise the stream could not be split up into equal partitions.

Initially, the scheduler is set to streaming mode. To switch to replication mode, you have to prepare the scheduler:

```
sched = Ufo.Scheduler(remotes=remotes)
sched.set_remote_mode(Ufo.RemoteMode.REPLICATE)
sched.run(graph)
```

Improving small kernel launches

UFO uses a single OpenCL context to manage multiple GPUs in a transparent way. For applications and plugins that require many small kernel launches, multi-GPU performance suffers on NVIDIA systems due to bad scaling of the kernel launch time. In order to improve performance on machines with multiple GPUs it is strongly advised to run multiple `ufod` services with differently chosen GPUs and ports.

JSON Configuration Format

JSON is a self-contained, human-readable data-interchange format. It is pure Unicode text and language independent. The main structures objects containing key/value pairs (hash-tables, dictionaries, associative arrays ...) and ordered lists (arrays, vectors, sequences ...) of objects or values. For a complete description you may refer to the complete reference at json.org.

The configuration of a filter setup is stored in a JSON-encoded text file with a `.json` suffix. The root object must at least contain a `nodes` and an `edges` array

```
{ "nodes": [], "edges": [] }
```

The root object may also define several `prop-set` s for further reference.

Nodes array

The nodes array contains filter objects that are executed on run-time. Information how they are connected is provided in the *Edges array*.

Filter object

A filter consists at least of a `plugin` key string pointing to the filter that is going to be used and a `name` string field for unique identification. Of course, plugins have to be available as a shared object in UFO's path.

To configure the filter, the `properties` field can be used. This is an object that maps string keys specifying the actual filter property to the value. Therefore, the Python code to set a property

```
reader = graph.get_filter('reader')
reader.set_properties(path='/home/user/data/*.tif', count=5)
```

translates to

```
{ "path": "/home/user/data/*.tif", "count": 5 }
```

Instead of defining recurring properties for each filter, you can also use pre-defined *Property sets*.

Example nodes array

An example node array looks like this:

```
"nodes" : [
    {
        "plugin": "reader",
        "name": "reader",
        "properties" : { "path": "/home/user/data/*.tif", "count": 5 }
    },
    {
        "plugin": "writer",
        "name": "writer"
    }
]
```

Edges array

The edges array specifies how the nodes in a *Nodes array* are connected. Each entry is an object that contains two objects `from` and `to`. In both objects you have to specify at least the node name with the `name` key. Furthermore, if there are several inputs or outputs on a node, you have to tell which input and output to use with the `input` on the `to` node and the `output` key on the `from` node. If you omit these, they are assumed to be 0.

To connect the nodes defined in the *Example nodes array* all you have to do is

```
"edges" : [
    {
        "from": {"name": "reader"},
        "to": {"name": "writer", "input": 2}
    }
]
```

Note, that the names specify the name of the node, not the plugin.

Property sets

To avoid to list the same properties for different filters over and over again, properties can be pre-defined with a singular top-level `prop-sets` object. Each key is a name that can be referenced in filter nodes using the `prop-refs` array. The values are ordinary `property` mappings:

```
{
    "prop-sets" : {
        "foo-prop": {
            "path": "/home/user/path/to/projections/*.tif",
            "count": 300
        }
    },
    "nodes" : [
        {
            "plugin": "reader",
            "name": "reader",
            "prop-refs": ["foo-prop"]
        }
    ]
}
```

Loading and Saving the Graph

The `UfoGraph` class exports the `ufo_graph_read_from_json` and `ufo_graph_read_save_to_json` methods which are responsible for loading and saving the graph. In Python this would simply be:

```
from gi.repository import Ufo

g1 = Ufo.Graph()

# set up the filters using graph.get_filter() and filter.connect_to()

g1.run()
g1.save_to_json('graph.json')

g2 = Ufo.Graph()
g2.load_from_json('graph.json')
g2.run()
```

Developing new task filters

UFO filters are simple shared objects that expose their `GType` and implement the `UfoFilter` class.

Writing a task in C

Writing a new UFO filter consists of filling out a pre-defined class structure. To avoid writing the GObject boild plate code, you can call

```
ufo-mkfilter AwesomeFoo
```

to generate the header and source file templates. The name must be a camel-cased version of your new filter.

You are now left with two files `ufo-awesome-foo-task.c` and `ufo-awesome-foo-task.h`. If you intend to distribute that filter with the main UFO filter distribution, copy these files to `ufo-filters/src`. If you are not depending on any third-party library you can just add the following line to the `CMakeLists.txt` file:

```
set(ufofilter_SRCS
    ...
    ufo-awesome-foo-task.c
    ...)
```

You can compile this as usual by typing

```
make
```

in the CMake build directory of `ufo-filters`.

Initializing filters

Regardless of filter type, the `ufo_awesome_foo_task_init()` method is the *constructor* of the filter object and the best place to setup all data that does not depend on any input data.

The `ufo__awesome_foo_task_class_init()` method on the other hand ist the *class constructor* that is used to *override* virtual methods by setting function pointers in each classes' vtable.

You *must* override the following methods: `ufo_awesome_task_get_num_inputs`, `ufo_awesome_task_get_num_dimensions`, `ufo_awesome_task_get_mode`, `ufo_awesome_task_setup`, `ufo_awesome_task_get_requisition`, `ufo_awesome_task_process` and/or `ufo_awesome_task_generate`.

`get_num_inputs`, `get_num_dimensions` and `get_mode` are called by the run-time in order to determine how many inputs your task expects, which dimensions are allowed on each input and what processing mode your task runs

```
static guint
ufo_awesome_task_get_num_inputs (UfoTask *task)
{
    return 1;   /* We expect only one input */
}

static guint
ufo_awesome_task_get_num_dimensions (UfoTask *task, guint input)
{
    return 2;   /* We ignore "input" and always expect 2 dimensions */
}

static UfoTaskMode
ufo_awesome_task_get_mode (UfoTask *task)
{
    /* We process one item after another */
    return UFO_TASK_MODE_PROCESSOR;
}
```

The mode decides which functions of a task are called. Each task can provide a `process` function that takes input data and optionally writes output data and a `generate` function that does not take input data but writes data. Both functions return a boolean value to signal if data was produced or not (e.g. end of stream):

- `UFO_TASK_MODE_PROCESSOR`: The task reads data and optionally writes data. For that it must implement `process`.

- `UFO_TASK_MODE_GENERATOR`: The task only produces data (e.g. file readers) and must implement `generate`.
- `UFO_TASK_MODE_REDUCTOR`: The tasks reads the input stream and produces another output stream. Reading is accomplished by implementing `process` whereas production is done by `generate`.

`setup` can be used to initialize data that depends on run-time resources like OpenCL contexts etc. This method is called only *once*

```
static void
ufo_awesome_task_setup (UfoTask *task,
                        UfoResources *resources,
                        GError **error)
{
    cl_context context;

    context = ufo_resources_get_context (resources);

    /*
       Do something with the context like allocating buffers or create
       kernels.
     */
}
```

On the other hand, `get_requisition` is called on each iteration right before `process`. It is used to determine which size an output buffer must have depending on the inputs. For this you must fill in the `requisition` structure correctly. If our output buffer needs to be as big as our input buffer we would specify

```
static void
ufo_awesome_task_get_requisition (UfoTask *task,
                                  UfoBuffer **inputs,
                                  UfoRequisition *requisition)
{
    ufo_buffer_get_requisition (inputs[0], requisition);
}
```

Finally, you have to override the `process` method

```
static gboolean
ufo_awesome_task_process (UfoTask *task,
                          UfoBuffer **inputs,
                          UfoBuffer *output,
                          UfoRequisition *requisition)
{
    UfoGpuNode *node;
    cl_command_queue cmd_queue;
    cl_mem host_in;
    cl_mem host_out;

    /* We have to know to which GPU device we are assigned to */
    node = UFO_GPU_NODE (ufo_task_node_get_proc_node (UFO_TASK_NODE (task)));

    /* Now, we can get the command queue */
    cmd_queue = ufo_gpu_node_get_cmd_queue (node);

    /* ... and get hold of the data */
    host_in = ufo_buffer_get_device_array (inputs[0], cmd_queue);
    host_out = ufo_buffer_get_device_array (output, cmd_queue);
```

```
    /* Call a kernel or do other meaningful work. */
}
```

Tasks can and will be copied to speed up the computation on multi-GPU systems. Any parameters that are accessible from the outside via a property are automatically copied by the run-time system. To copy private data that is only visible at the file scope, you have to override the `UFO_NODE_CLASS` method `copy` and copy the data yourself. This method is *always* called before `setup` so you can be assured to re-create your private data on the copied task.

Note: It is strongly encouraged that you export all your parameters as properties and re-build any internal data structures off of these parameters.

Additional source files

For modularity reasons, you might want to split your filter sources into different compilation units. In order to compile and link them against the correct library, add the following statements to the `src/CMakeLists.txt` file

```
set(awesome_foo_misc_SRCS foo.c bar.c baz.c)
```

in case your filter is still called `AwesomeFoo`. Notice, that the variable name matches the plugin name with underscores between the lower-cased letters.

Writing point-based OpenCL filters

For point-based image operations it is much faster to use the cl-plugin that writing a full-fledged C filter. We create a new file `simple.cl`, that contains a simple kernel that inverts our normalized input (you can silently ignore the `scratch` parameter for now):

```
kernel void invert(global float *input, global float *output)
{
    /* where are we? */
    int index = get_global_id(1) * get_global_size(0) + get_global_id(0);
    output[index] = 1.0f - input[index];
}
```

We wire this small kernel into this short Python script:

```
from gi.repository import Ufo

pm = Ufo.PluginManager()
reader = pm.get_filter('reader')
writer = pm.get_filter('writer')

# this filter applies the kernel
cl = pm.get_filter('opencl')
cl.set_properties(filename='simple.cl', kernel='invert')

g = Ufo.TaskGraph()
g.connect_nodes(reader, cl)
g.connect_nodes(cl, writer)

s = Ufo.Scheduler()
s.run(g)
```

For more information on how to write OpenCL kernels, consult the official OpenCL reference pages.

Reporting errors at run-time

From within a filter (or any library for that matter) never call functions such as `exit()` or `abort()`. This prevents the calling application from identification of the problem as well as recovery. Instead use the builtin `GError` infrastructure that – as a bonus – map nicely to exceptions in Python:

```
static void
ufo_awesome_task_setup (UfoTask *task,
                        UfoResources *resources,
                        GError **error)
{
    if (error_condition) {
        g_set_error (error, UFO_TASK_ERROR, UFO_TASK_ERROR_SETUP,
                     "Error because of condition");
        return;
    }
}
```

Note that `g_set_error` receives printf-style format strings which means you should be as specific as possible with the given error message.

The GObject property system

Wait until a property satisfies a condition

For some filters it could be important to not only wait until input buffers arrive but also properties change their values. For example, the back-projection should only start as soon as it is assigned a correct center-of-rotation. To implement this, we have to define a condition function that checks if a `GValue` representing the current property satisfies a certain condition

```
static gboolean is_larger_than_zero(GValue *value, gpointer user_data)
{
    return g_value_get_float(value) > 0.0f;
}
```

As the filter installed the properties it also knows which type it is and which `g_value_get_*()` function to call. Now, we wait until this conditions holds using `ufo_filter_wait_until()`

```
/* Somewhere in ufo_filter_process() */
ufo_filter_wait_until(self, properties[PROP_CENTER_OF_ROTATION],
        &is_larger_than_zero, NULL);
```

> **Warning:** `ufo_filter_wait_until()` might block indefinitely when the condition function never returns `TRUE`.

See also:

How can I synchronize two properties?

Frequently Asked Questions

Installation

Why can't the linker find libufo.so?

In the rare circumstances that you installed UFO from source for the first time by calling `make install`, the dynamic linker does not know that the library exists. If this is the case issue

```
$ sudo ldconfig
```

on Debian systems or

```
$ su
$ ldconfig
```

on openSUSE systems.

If this is not working, the library is neither installed into `/usr/lib` nor `/usr/local/lib` on 32-bit systems or `/usr/lib64` and `/usr/local/lib64` on 64-bit systems.

Usage

Why do I get a "libfilter<foo>.so not found" message?

Because the UFO core system is unable to locate the filters. By default it looks into `${LIBDIR}/ufo`. If you don't want to install the filters system-wide, you can tell the system to try other paths as well by appending paths to the `UFO_PLUGIN_PATH` *environment variable*.

Can I split a linear data stream?

The output data stream of a node can be split by setting the `UFO_SEND_SEQUENTIAL` mode and adjusting the number of expecting data items on each connected node:

```
from gi.repository import Ufo

out_node.set_send_pattern(Ufo.SendPattern.SEQUENTIAL)
in1_node.set_num_expected(0, 5)  # expect five items on the first input
in2_node.set_num_expected(0, -1) # expect all items

g = Ufo.TaskGraph()
g.connect_nodes(out_node, in1_node)
g.connect_nodes(out_node, in2_node)
```

The connection order matters here! If it would be reversed, `in2_node` would receive all items whereas `in1_node` wouldn't receive anything.

How can I control the debug output from libufo?

Generally, UFO emits debug messages under the log domain `Ufo`. If you use a UFO-based tool and cannot see debug messages, you might have to enable them by setting the `G_MESSAGES_DEBUG` environment variable, i.e.:

```
export G_MESSAGES_DEBUG=Ufo
```

To handle these messages from within a script or program, you must set a log handler that decides what to do with the messages. To ignore all messages in Python, you would have to write something like this:

```
from gi.repository import Ufo, GLib

def ignore_message(domain, level, message, user):
    pass

if __name__ == '__main__':
    GLib.log_set_handler("Ufo", GLib.LogLevelFlags.LEVEL_MASK,
        ignore_message, None)
```

How can I use Numpy output?

Install the `ufo-python-tools`. You can then use the BufferInput filter to process Numpy arrays data:

```
from gi.repository import Ufo
import ufo.numpy
import numpy as np

arrays = [ i*np.eye(100, dtype=np.float32) for i in range(1, 10) ]
buffers = [ ufo.numpy.fromarray(a) for a in arrays ]

pm = Ufo.PluginManager()
numpy_input = pm.get_task('bufferinput')
numpy_input.set_properties(buffers=buffers)
```

How can I synchronize two properties?

Although this is a general GObject question, synchronizing two properties is particularly important if the receiving filter depends on a changed property. For example, the back-projection should start only if a center-of-rotation is known. In Python you can use the `bind_property` function from the `ufotools` module like this:

```
from gi.repository import Ufo
import ufotools.bind_property

pm = Ufo.PluginManager()
cor = g.get_task('centerofrotation')
bp = g.get_task('backproject')

# Now connect the properties
ufotools.bind_property(cor, 'center', bp, 'axis-pos')
```

In C, the similar `g_object_bind_property` function is provided out-of-the-box.

CHAPTER 2

Developers

Reporting Bugs

Bug reports regarding the UFO framework should be submitted to the ufo-core issue tracker. Anything related to specific filter behaviour should be filed on the ufo-filters issue tracker.

Changelog

Here you can see the full list of changes between each ufo-core release.

Version 0.13.0

Released on January 25th 2017.

Enhancements:

- ufo-runjson: Add the -s/–scheduler flag to choose a scheduler different from the regular one.
- Restructure docs and add section about broadcasts
- Added ufo_resources_get_kernel_source function to have an API to access directly a source file from CL/kernel path.

Fixes:

- Fixup for compilation and installation on MacOS

Version 0.12.1

Bugfix release released on November 28th 2016.

- Do not install the Docbook XML build dir

- Document UFO_BUFFER_DEPTH_INVALID
- Enable Large File Support
- Check ftell and return NULL on error
- Add forgotten manpage to the build list
- Initialize uninitialized variable
- Enable _FORTIFY_SOURCE feature flag

Version 0.12.0

Released on November 24th 2016.

Enhancements:

- ufo-launch: convert string to enum values
- ufo-runjson: add -t/–trace flag doing the same thing as ufo-launch
- Documentation updates

Fixes:

- Fix manual heading
- ufo-mkfilter: fix template and type handling
- Add manpage for ufo-prof

Breaks:

- Add UFO_BUFFER_DEPTH_INVALID with value zero which means adding to the API and breaking ABI
- Remove package target from the build system

Version 0.11.1

Bugfix release released on November 12th 2016.

- Install systemd unit file through pkg-config
- Fix #110: install templates correctly
- Do not run xmllint on manpage generation output
- Remove unused CMake modules
- Remove executable bit from source files
- Remove PACKAGE_* variables

Version 0.11

Released on November 8th 2016

Enhancements:

- Build manpages for the tools
- Update TomoPy integration

- Add UFO_DEVICE_TYPE environment variable
- Properly build on MacOS
- Unify debug message output format
- ufo-launch: rewrite specification parser allowing more flexible descriptions
- ufo-launch: add –quieter
- ufo-mkfilter: add –type and –use-gpu

Fixes:

- Check if we have with multiple roots to exit early
- Show version information consistently
- ufo-prof: support Python < 3.0

Breaks:

- GNUInstallDirs instead is now used instead of our own ConfigurePaths CMake module which might affect installation paths on your system. Please note that for example -DPREFIX thus becomes -DCMAKE_INSTALL_PREFIX.
- Remove CPack
- ufod: unused –paths option removed
- ufo-runjson: unused –path option removed
- ufo-launch: do not execute graph when –dump'ing

Version 0.10

Released on May 24th 2016

Enhancements:

- Add UFO_GPU_NODE_INFO_MAX_MEM_ALLOC_SIZE
- Fix #103: allow ufo-launch to use arbitrary graphs with new workflow specification and more robust parsing
- Look for AMD APP SDK 3.0
- Cache programs to avoid rebuilding them
- ufo-runjson now also outputs the number of processed items
- Fix #104: output type and blurb with -v

Fixes:

- Fix #106: match word characters to find plugins
- Fix #101: really unref non-intermediate nodes
- ufo-launch: query only valid graph
- Fixed misleading documentation
- Fixed some leaks and unreferenced resources
- Stop num-processed from being serialized into JSON field
- ufod: fix segfault if no address is specified

Breaks:

- ufo-query: print errors on stderr instead of stdout
- ufo-launch: removed –progress and –time in favor of –quiet
- Removed public ufo_signal_emit symbol
- Use date and time to differentiate trace profiles
- Replace clprof with ufo-prof

Version 0.9.1

Release on January 12th 2016.

Enhancements:

- Pass a property map to scheduler in the Python wrapper
- Added convenience wrapper to copy into a buffer

Fixes:

- Fix version numbering from 0.8.x to 0.9.x
- Fix documentation issues
- Fix build problems if libzmq is not present
- ufo-launch unrefs non-intermediate nodes

Version 0.9

Release on November 3rd 2015.

Enhancements:

- Generally improved debug output
- Add support for plugin packages as demonstrated by the ART plugins
- Add Docker installation method
- Add simple fabfile to start ufod instances
- Allow reductors to pause processing
- Add ufo-query binary to retrieve info about tasks
- Add ufo_buffer_set_device_array API call
- Python: add ufo.numpy.empty_like
- ufo-launch: add –dump flag to serialize to JSON
- ufo-launch: add Bash completion script
- ufo-launch: pass address list

Fixes:

- Fix segfault with long-running tasks
- Prevent daemon from leaking OpenCL resources
- Fix broken continuous daemon operation
- ufo-launch: fix parsing uint64

- Use same reductor policy for both schedulers
- Fix ufo_buffer_new_with_data
- Fix nano second to second conversion

Breaks:

- Specify device subset with UFO_DEVICES
- Do not prepend . to trace and profile output
- Change ufo_buffer_set_host_array signature
- Use common timestamp unit for both trace types

Version 0.8

Release on May 19th 2015.

This release breaks with the distinction of ArchGraph and Resources. The former is removed with its functionality moved to the latter. Besides that numerous improvements have been incorporated:

- ufo-launch: add –time and –trace options
- Added a systemd unit for ufod
- Fixed cluster communication.
- Fixed #86: use CL_INTENSITY instead of CL_R
- Handle kernel path with environment variable UFO_KERNEL_PATH
- ufo-launch now parses boolean properties
- Removed hard dependency on ZeroMQ
- Removed WITH_DEBUG and WITH_PROFILING CMake variables
- Add JSON-based TANGO server for remote computation
- Removed ufo_base_scheduler_get_gpu_nodes()
- Fixed #80: add ufo_gpu_node_get_info()
- Fixed re-running the same task graph
- Removed tifffile dependency
- Fixed #78: reset num-processed before execution
- Fixed #77: initialize threads when accessing the GIL

Version 0.7

Released on February 20th 2015.

This is a major update from the previous version released a year ago. Besides numerous bug fixes and compatibility enhancements we

- added ufo_signal_emit for threadsafe signal emission
- export plugin and kernel directories via pkg-config so that thirdparty plugins easily know where to install themselves
- added the ufo-launch tool to run basic pipelines directly from the command line

- merged the ufo-python-tools and ufo-tests repos with the core repository
- added additional buffer depths
- parse the UFO_PLUGIN_PATH environment variable to specify additional plugin locations
- added a copy task which copies data from input 0
- added a fixed scheduler for manual assignment of hardware resources to task nodes
- added "processed" and "generated" signals to tasks which emit whenever either action completed
- added UFO_DEVICE_TYPE_ACC for accelerator devices
- added buffer views for larger-than-GPU data
- output OpenCL profiler information as Chrome JSON
- added ufo_resources_get_kernel_with_opts()
- added ufo_buffer_set_host_array()
- added ufo_buffer_get/set_metadata(), ufo_buffer_get_metadata_keys() and ufo_buffer_copy_meta_data()
- added ufo_buffer_get_device_array_with_offset()
- added ufo_buffer_get_location()

and broke compatibility by

- retiring the UfoConfig infrastructure and
- replacing g_message() with g_debug()

Version 0.6

Released on January 24th 2014.

Due to the inclusion of Autotools builds, we restructured and cleaned up installation paths, along the lines of GLib. That means headers are installed into major API version dependent directories, e.g. now /usr/include/ufo-0/ufo. We also split hardware-dependent and -independent files, thus kernels go into /usr/share/ufo now. In the same vein, we also lowered the SO version down to 0, so please if you have installed from source, make sure to remove any traces of an old UFO installation before installing the latest version.

Minor changes include:

- Added "time" property to scheduler.
- Scheduler can now use and existing arch graph and return associated resources.
- By default only GPU devices are used.
- Documentation is now hosted at ufo-core.readthedocs.org.
- Added "num-processed" property.

Developers

- Replace GList for loops with g_list_for macro
- Include compat.h
- Renamed and install mkfilter as ufo-mkfilter.
- ufo-core is now monitored by Travis CI

- Removed the python/ subdir which is replaced by ufo-python-tools
- Removed unnecessary clprof tool.

Bugfixes

- Fix and simplify deploy script
- Require json-glib in pkgconfig
- Link to documentation on rtd.org
- Fix .gir generation
- Fix installation path for header files
- Do not error on deprecated declarations

Version 0.5

Released on October 28th 2013.

- Added MPI support as an alternative to ZMQ.
- Added basic math operations for use with filters.
- UFO can now be used reliably in a multithreaded Python context. That means, calling `ufo_scheduler_run` in a Python thread will "just work". This change allows run-time injection of NumPy buffers into the task graph.

Developers

- Add `-DDEBUG` when debug is enabled so we can #ifdef it
- Add GLib version guards
- ufo-core compiles with Clang
- CMake 2.6 is used solely throughout the sources
- Add convenience function ufo_buffer_new_with_size
- Add a shim macros to support both zmq 2 and 3
- Add `UFO_USE_GPU` env var to restrict to single GPU
- Added ufo_resource_manager_get_cached_kernel that always returns the same kernel object when given the same file and kernel name. Note, that you have to guard it properly and do not call `clSetKernelArg` from multiple threads at the same time.
- Add profile tracing to produce a JSON trace event file, that can be read and visualized with Google's Chrome browser. It can be enabled with UfoScheduler::enable-tracing set to TRUE.

Bug fixes

- Fix #6: Don't use enum values as bit flags
- Fix bug: no plugin name is sent to remote nodes
- Fix copy segfault in when source has not alloc'd
- Removed dependency on a C++ compiler

- Fix reduction problem

Version 0.4

Released on July 18th 2013.

Major changes

- Rewrote internal architecture for better scheduling.
- Remove profiler levels and add more output
- Implement input data partitioning: On clusters where distributed data access is possible, we can achieve perfect linear scalability by partitioning the input data set.
- Install SIGTERM handler for cleanup of node server

Features

- Add ufo_task_graph_get_json_data
- Streamline and simplify scheduling
- Provide function to flatten graph
- Provide graph copy functionality
- Add node indices for copies
- Add all paths as OpenCL include paths
- Write out JSON version
- Search in UFO_PLUGIN_PATH env var

Bug fixes

- Fix problems with AMD platforms
- Fix timestamp readout
- Fix potential single integer overflow
- Exit when JSON tasks could not be found
- Fix remote tasks getting stuck
- Unref expanded nodes explicitly
- Fix #189: don't copy nodes with more than one input
- Fix #219: Warn instead of segfault
- Fix annotation for older GI compiler
- Fix problem with first remote data item
- Fix platform selection
- Fix problems with objects that are not unreffed
- Refactor buffer and add support for #184

- Refactor resources and fix #183
- Fix buffer for broadcast operations

Version 0.3

Released on February 8th 2013.

Major breakage

- A graph is now a simple data structure and not a specific graph of task nodes. This is implemented by a TaskGraph.
- Filters are now called TaskNodes and connected with `ufo_task_graph_connect_nodes` and `ufo_task_graph_connect_nodes_full` respectively.

Graph expansion

With 0.2, Using multiple GPUs was possible by manually splitting paths in the graph and assigning GPUs. Now, task graphs are automatically expanded depending on the number of available GPUs and remote processing slaves that are started with the newly added `ufod` server software.

Minor improvements

- A `deploy.sh` script has been added for easier deployment of the software stack. This is especially useful to install everything in the home directory of the user, who only needs to setup `LD_LIBRARY_PATH` and `GI_TYPELIB_PATH` correctly to run the software.

Version 0.2

Released on November 8th 2012.

Major breakage

- Filters are now prefixed again with `libfilter` to allow introspected documentation. Thus, any filter built for 0.1 cannot be used because they are simply not found.
- `ufo_plugin_manager_get_filter()` received a new third parameter `error` that reports errors when opening and loading a UfoFilter from a shared object.
- `ufo_resource_manager_add_program()` is removed.
- The kernel file name must be passed to `ufo_resource_manager_get_kernel()`.
- The `CHECK_ERROR` macro defined in `ufo-resource-manager.h` was renamed to `CHECK_OPENCL_ERROR` to better reflect its real purpose.
- The old JSON specification has been changed to reflect the possibilities of the current API. Thus, JSON files that worked under Ufo 0.1 cannot be read with Ufo 0.2.
- Removed the otherwise unused `ufo_buffer_get_transfer_time()` and replaced this with the more flexible `ufo_buffer_get_transfer_timer()`.

- Rename `ufo_filter_initialize()` to `ufo_filter_set_plugin_name()` that reflects its true meaning.

Scheduling

A more scheduable way to run filters has been implemented with the virtual `process_cpu()` and `process_gpu()` methods. Contrary to the old way, they act on *one* working set at a time that is passed as an array of pointers to `UfoBuffer`. Sometimes, a filter needs to setup data depending on the input size. For this reason, the virtual method `initialize()` takes a second parameter that is again a list of pointers to buffer objects.

Moreover, the `UfoScheduler` class has been added that is combining the work previously accomplished by `ufo_filter_process()` and `ufo_graph_run()`. The scheduler orchestrates the filters and assigns resources in a meaningful way.

If written in the new kernel style, producer filters must return a boolean flag denoting if data was produced or not.

General improvements

- The manual was restructured considerably.
- Saving graphs as JSON files has been added via `ufo_graph_save_to_json()`.
- Filters can now wait until their properties satisfy a condition using `ufo_filter_wait_until()`, see also *Wait until a property satisfies a condition*.
- A new method `ufo_resource_manager_get_kernel_from_source()` so that filters can load kernels directly from source.
- Streamlined error handling: Filters should not issue `g_warnings` or `g_errors` on their own anymore but create an error with `g_error_new` and return that.

Version 0.1.1

- Ticket #55: tests/test-channel blocks indefinitely

CHAPTER 3

Additional notes

Copyright

Index

E