Karlsruhe Institute of Technology

Gentle introduction to the UFO ecosystem

Matthias Vogelgesang

matthias.vogelgesang@kit.edu

Institute for Data Processing and Electronics

SET UP

Preliminaries

OpenCL

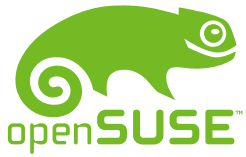
- NVIDIA OpenCL part of CUDA for NVIDIA GPUs
- AMD SDK for AMD GPUs, APUs and Intel CPUs
- Intel SDK for Intel CPUs and Xeon Phi

Manual installation

- OpenCL headers (*must* fit target hardware, i.e. 1.1 for NVIDIA)
- CMake
- GLib >= 2.30 (libglib-2.0-dev)
- json-glib (libjson-glib-dev)
- Optional: gobject-introspection, python-numpy

Pre-built packages for openSUSE

- `zypper ar repo-ufo http://download.opensuse.org/repositories/home:/ufo-kit/openSUSE_x.y/`
- `zypper in ufo-core`
- `zypper in ufo-filters`
- Optional: `zypper in python-ufo-tools`



Manual installation

Fetch sources

- `git clone https://github.com/ufo-kit/ufo-{core,filters}`

Configure and build

- `cd ufo-{core,filters}`
- `cmake . -DPREFIX=<install-prefix> -DLIBDIR=<lib-install-dir>`
- `make && make install`

Just for today ...

- `sudo apt-get install python-numpy python-gobject-dev python-setuptools`
- `cd ufo-core/python && python setup.py install --user`

USAGE

Workflow

User side

- Instantiate and configure tasks
- Connect tasks in a pipeline/graph
- Execute using a scheduler

Behind the scenes

- Scheduler determines hardware setup (i.e. CPUs, GPUs, remote machines)
- Transform the input graph to match the hardware
- Assign each task a hardware resource
- In a loop fetch and scatter data

Command line interface

Karlsruhe Institute of Technology

DEMO

Low-level C – explicit

```
UfoPluginManager *pm;
UfoTaskGraph *graph;
UfoScheduler *sched;

pm = ufo_plugin_manager_new ();
reader = ufo_plugin_manager_get_task (pm, "read");
writer = ufo_plugin_manager_get_task (pm, "write");

g_object_set (reader, "path", "*.tif", NULL);
ufo_task_graph_connect (reader, writer);

ufo_scheduler_run (sched, graph);

g_object_unref (pm);
```

Low-level C – data-driven

```
UfoPluginManager *pm;
UfoTaskGraph *graph;
UfoScheduler *sched;
GError *error;

/* Load from JSON file */
ufo_task_graph_read_from_file (graph, pm, "input.json", &error);
ufo_scheduler_run (sched, graph);

/* Save to JSON */
ufo_task_graph_save_to_json (graph, "foo.json");
```

Low-level Python

```
from gi.repository import Ufo

pm = Ufo.PluginManager()
graph = Ufo.TaskGraph()
sched = Ufo.Scheduler()

read = pm.get_task('read')
write = pm.get_task('write')
read.set_properties(path='*.tif')
write.set_properties(filename='output.h5')

graph.connect_nodes(read, write)
sched.run(graph)
```

High-level Python

```
import numpy
from ufo import Read, Write, Backproject

read = Read(path='*.tif')
write = Write(filename='output.h5')
backproject = Backproject(axis_pos=512.0)

# run and wait for completion
write(read()).run.join()

# or using the result as a generator
for item in backproject(read()):
    print("mean: {}".format(numpy.mean(item)))
```

Usage summary

Low-level C summary

+ C API allows strong interoperability
+ Low overhead
- May be awkward to use

High-level Python summary

+ Easy to use
+ Fast prototyping
- Less control

Low-level Python summary

+ High degree of control
- Still requires some work

It depends what you want to do …

EXTENSION

GPU computing in a nutshell

Execution model

- A kernel is executed by work items on a *n*-dimensional index space
- Kernel uses its index to address data or task
- Speed-up caused by massive parallelism (e.g. one pixel operation per kernel)
- Synchronization at either kernel or work group level

Programming model

- Host uses OpenCL C run-time API for hardware communication
- Kernel is written in a subset of C99 that is compiled at run-time
- Data transfers must be initiated explicitly

OpenCL point operations

```
source = r"""
kernel void
binarize(global float* input,
         global float* output)
{
  int idx = get_global_id(1) * get_global_size(0) +
            get_global_id(0);
  output[idx] = input[idx] > 256.0f ? 1.0f : 0.0f;
}"""

from ufo import Opencl
binarize = Opencl(source=source)
```

OpenCL point operations using transformation

```
from pina import static

@static
def binarize(x):
  return 1.0 if x > 256.0 else 0.0

task = Opencl(source=binarize)
```

OpenCL point operations as expressions

```
from ufo import Calculate
binarize = Calculate(expression='v > 256 ? 1 : 0')
```

Completely new tasks

Karlsruhe Institute of Technology

Why?

- Integration of third-party libraries
- Custom setup required
- Temporary memory or pre-computated tables

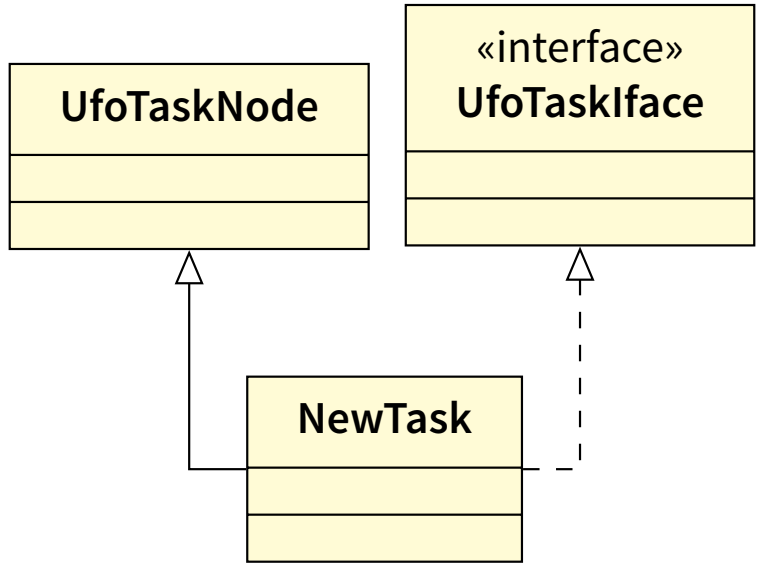
Completely new tasks

Why?

- Integration of third-party libraries
- Custom setup required
- Temporary memory or pre-computated tables

Basic procedure

- Derive a new class from `UfoTaskNode`
- Implement `UfoTaskIface` interface



Task implementation

GObject inheritance

- OO in C using structs and manual vtable
- Requires some boiler plate → `ufo-mkfilter`

Event loop between scheduler and task

1. Setup task and query basic info
2. While not finished
   - 2.1 Send inputs and ask for size requirements
   - 2.2 Send inputs and output and let task process
   - 2.3 Receive result
3. Free task's resources

Setup and static info

```c
static void
task_setup (UfoTask *task, UfoResources *r, GError **error)
{
    /* Use r to get OpenCL context etc. if neccessary */
}

static guint
task_get_num_inputs (UfoTask *task)
{
    return 1;
}

static guint
task_get_num_dimensions (UfoTask *task, guint input)
{
    return 2;
}
```

Setup and static info cont'd

```
static UfoTaskMode
task_get_mode (UfoTask *task)
{
    return UFO_TASK_MODE_PROCESSOR | UFO_TASK_MODE_GPU;
}
```

Modes

- PROCESSOR: 1:1 processing
- GENERATOR: 0:m data generation
- REDUCTOR: m:n reduction
- SHARE_DATA: siblings receive the same input data

Per iteration

```
static void
task_get_requisition (UfoTask *task, UfoBuffer **inputs,
                      UfoRequisition *requisition)
{
  /* simple case: use same size as input */
  ufo_buffer_get_requisition (inputs[0], requisition);
}

static gboolean
task_process (UfoTask *task, UfoBuffer **inputs, UfoBuffer *output,
              UfoRequisition *requisition)
{
  /* use inputs to produce output */
  return TRUE;
}
```

Processing

One iteration

- Query associated resource with `ufo_task_node_get_proc_node()`
- Get command queue using `ufo_gpu_node_get_cmd_queue()`
- Get pointers for input and output data buffers
  - `ufo_buffer_get_host_array(inputs[0], NULL)`
  - `ufo_buffer_get_device_array(inputs[0], cmd_queue)`
  - `ufo_buffer_get_device_image(inputs[0], cmd_queue)`
- Run profiled call with `ufo_profiler_call` instead of raw `clEnqueueNDRangeKernel()`

Advantage

- “Outside” state hidden by run-time
- Task does not need to care where it runs and where data is located

Karlsruhe Institute of Technology

THANKS FOR YOUR ATTENTION