Karlsruhe Institute of Technology

Practical introduction to parallel computing with the UFO framework

Matthias Vogelgesang – Science 3D @ DESY

Institute for Data Processing and Electronics

Motivation

X-ray imaging

- Large spatial detector resolution (> 4 mega pixel)
- Large number of projections per scan (> 1000)
- Large number of scans

Data processing

- Reconstruction, pre- and post-processing of data *streams*
- Needs to be fast for quick assessment
- Easy to use and integrate

Solution

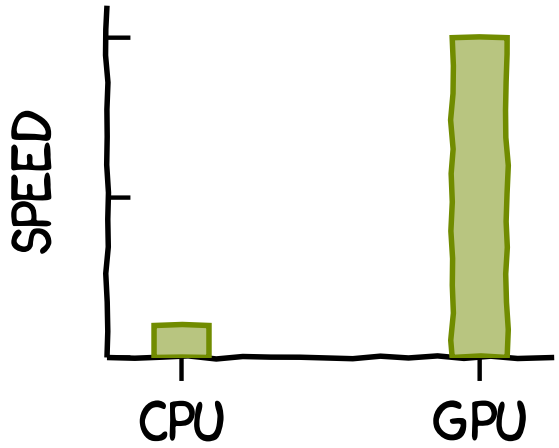
Hardware

- Parallel computing: increase throughput with multiple processors
- Heterogeneous computing: per-chip ISAs, co-processors, accelerators, …

Solution

Hardware

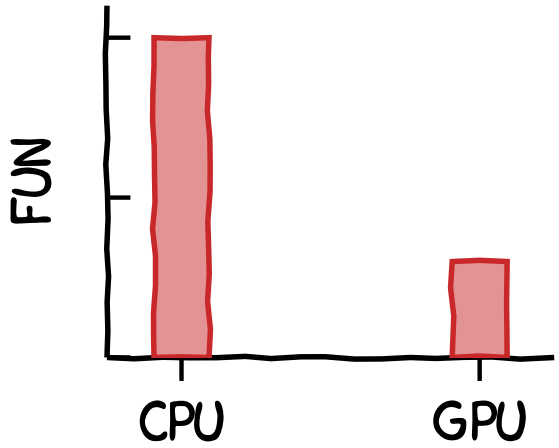
- Parallel computing: increase throughput with multiple processors
- Heterogeneous computing: per-chip ISAs, co-processors, accelerators, …



Solution

Hardware

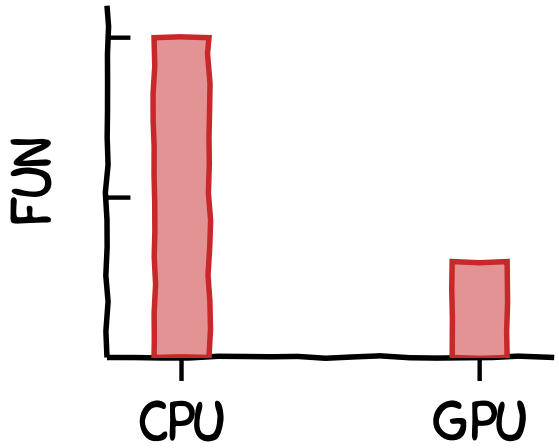
- Parallel computing: increase throughput with multiple processors
- Heterogeneous computing: per-chip ISAs, co-processors, accelerators, …
- However, programming parallel, heterogeneous systems is not trivial



Solution

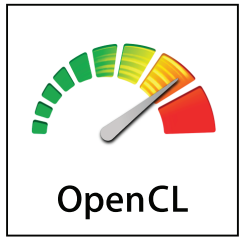
Hardware

- Parallel computing: increase throughput with multiple processors
- Heterogeneous computing: per-chip ISAs, co-processors, accelerators, …
- However, programming parallel, heterogeneous systems is not trivial



Software

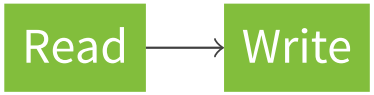
- Heterogeneous computing using OpenCL
- ufo framework abstracts low-level management
  - Execution on different devices
  - Data transfers and scheduling
  - Remote execution



Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O,



Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O, phase correction,

Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O, phase correction, sinogram,

Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O, phase correction, sinogram, filtered backprojection,

Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O, phase correction, sinogram, filtered backprojection, direct Fourier inversion

Split algorithm into simpler blocks

Graphs as algorithmic workflows

I/O, phase correction, sinogram, filtered backprojection, direct Fourier inversion , …

Installation

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

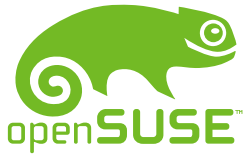
Manual installation

- OpenCL headers (*must* fit target hardware, i.e. 1.1 for NVIDIA)
- CMake
- GLib >= 2.22 (libglib-2.0-dev)
- json-glib (libjson-glib-dev)
- zeromq (libzmq-dev)
- Optional: gobject-introspection, python-numpy

Pre-built packages

openSUSE 12.2–13.1

- `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,python-tools}`

Configure and build

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

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

Low-level C – explicit

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

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

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

ufo_scheduler_run (sched, graph);

g_object_unref (pm);
/* ... */
```

Low-level C – data-driven

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

/* Load from JSON file */
ufo_task_graph_read_from_file (graph, pm,
    "description.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()

reader = pm.get_task('reader')
writer = pm.get_task('writer')
reader.set_properties(path='files/*.tif')

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

High-level Python

```
import numpy
from ufo import Reader, Writer, Backproject

read = Reader(path='files/*.tif')
write = Writer()
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)))
```

Note: requires Python tools

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 …

Extending

OpenCL tasks

```
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;
}"""

task = pm.get_task('opencl')
task.set_properties(source=source)

# ...
```

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 datum 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 tasks elegantly

```
from pina import static

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

task = pm.get_task('opencl')
task.set_properties(source=binarize)
```

Note: requires our Python-to-OpenCL compiler Piña

Completely new tasks

Why?

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

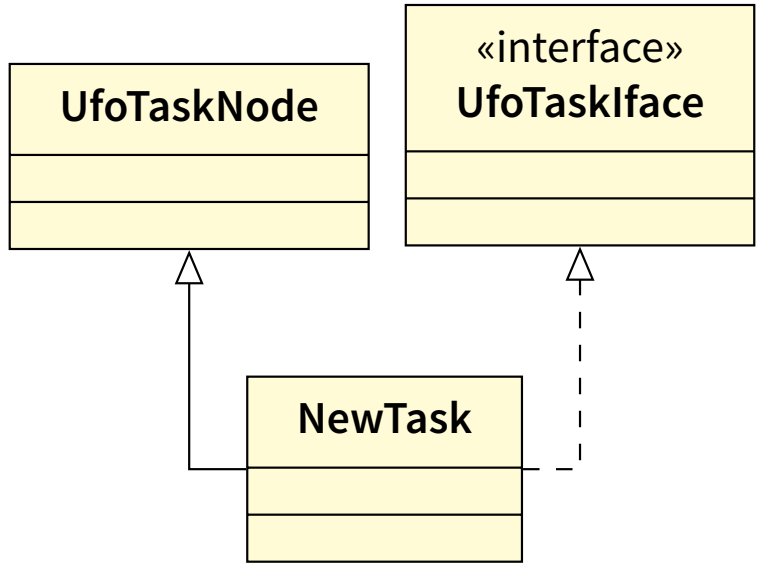
Completely new tasks

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

```
static void
task_setup (UfoTask *task, UfoResources *res, GError **error)
{
    /* Use res 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

Conclusion

Conclusion

Summary

- ufo framework relieves you from caring about hardware issues
- Interfaces nicely with Python
- Different levels of extensibility

More info

`ufo-kit/ufo-core` and `ufo.kit.edu`

Questions?