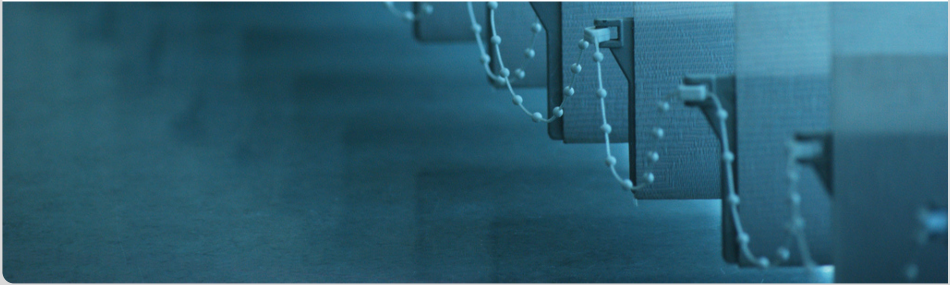
Karlsruhe Institute of Technology

Extensible Parallel Computing for Ultrafast X-Ray Imaging

Matthias Vogelgesang

Institute for Data Processing and Electronics



Outline

1. Introduction & Motivation

Outline

1. Introduction & Motivation
2. Technical Background

Outline

1. Introduction & Motivation
2. Technical Background
3. Results

Outline

1. Introduction & Motivation
2. Technical Background
3. Results
4. Usage

Situation

Large amounts of data

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560 $\times$ 2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs

Situation

KIT Karlsruhe Institute of Technology

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs
- ...but often unused due to lack of knowledge and manpower

Situation

Karlsruhe Institute of Technology

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs
- ...but often unused due to lack of knowledge and manpower

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs
- ...but often unused due to lack of knowledge and manpower

Increased requirements

Situation

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs
- ...but often unused due to lack of knowledge and manpower

Increased requirements

- On-site and on-line data processing (e.g. reconstruction)

Situation

Karlsruhe Institute of Technology

Large amounts of data

- High resolution detectors (e.g. pco.edge 2560×2160 at 16 Bits)
- Fast acquisition (100s of frames/sec)
- Automated sample changers

Excessive compute power

- Prevalance of multicore architectures and GPUs
- ...but often unused due to lack of knowledge and manpower

Increased requirements

- On-site and on-line data processing (e.g. reconstruction)
- Faster scans to „see“ dynamic processes

The UFO framework

UFO framework in a nutshell

The UFO framework

UFO framework in a nutshell

- Processes data streams (usually 1 to 4 dimensional floating point data)

The UFO framework

Karlsruhe Institute of Technology

UFO framework in a nutshell

- Processes data streams (usually 1 to 4 dimensional floating point data)
- Uses all hardware resources to increase throughput

The UFO framework

UFO framework in a nutshell

- Processes data streams (usually 1 to 4 dimensional floating point data)
- Uses all hardware resources to increase throughput
- Leverages GPU processing

The UFO framework

UFO framework in a nutshell

- Processes data streams (usually 1 to 4 dimensional floating point data)
- Uses all hardware resources to increase throughput
- Leverages GPU processing
- Hides parallelization and concurrency details

The UFO framework

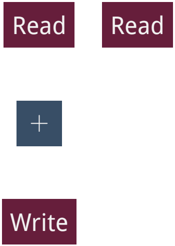
UFO framework in a nutshell

- Processes data streams (usually 1 to 4 dimensional floating point data)
- Uses all hardware resources to increase throughput
- Leverages GPU processing
- Hides parallelization and concurrency details
- Accessed via simple end-user interface

Task Graph Abstraction

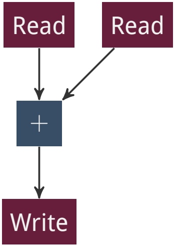
Task Graph Abstraction

- Define algorithms as self-contained tasks



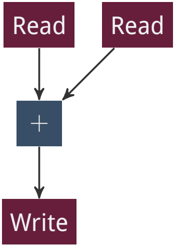
Task Graph Abstraction

- Define algorithms as self-contained tasks
- Specify data flow as edges in a graph



Task Graph Abstraction

- Define algorithms as self-contained tasks
- Specify data flow as edges in a graph
- Map the tasks to processing units such as CPU cores and GPUs

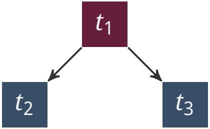


Benefits of graph task decomposition

- Tasks only depend on internal state and incoming data

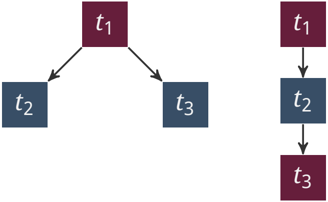
Benefits of graph task decomposition

- Tasks only depend on internal state and incoming data
- Sibling tasks can run in parallel

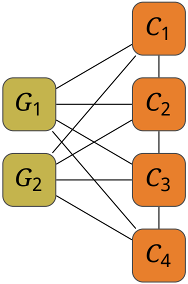


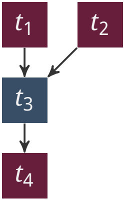
Benefits of graph task decomposition

- Tasks only depend on internal state and incoming data
- Sibling tasks can run in parallel
- Tasks in sequences can run in pipelined mode



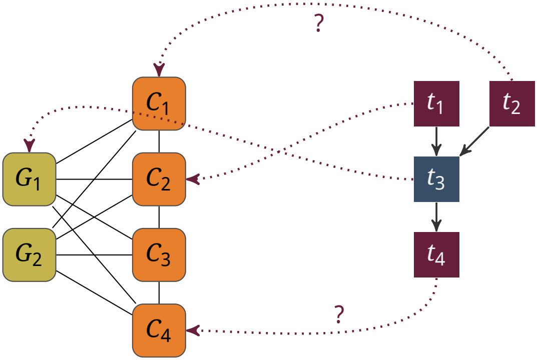
Scheduling challenges





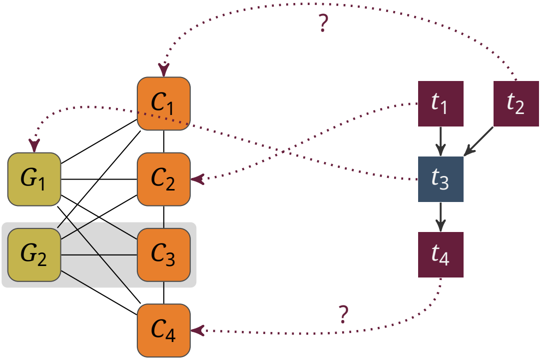
Scheduling challenges

- How to map tasks to processing units?



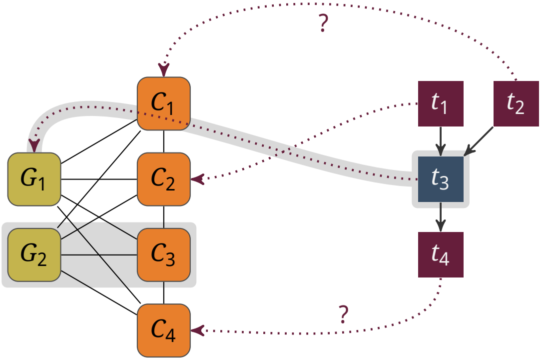
Scheduling challenges

- How to map tasks to processing units?
- How to use **all** processing units?



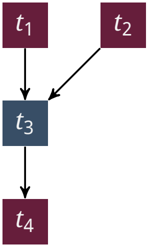
Scheduling challenges

- How to map tasks to processing units?
- How to use **all** processing units?
- What about hardware-specific tasks?



Graph expansion

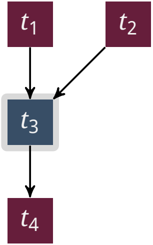
Using all processing units



Graph expansion

Using all processing units

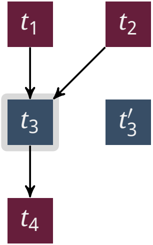
1. Breadth-first search to find **chains** of tasks that run on GPUs



Graph expansion

Using all processing units

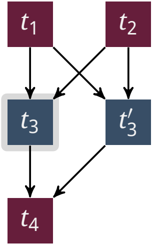
1. Breadth-first search to find **chains** of tasks that run on GPUs
2. Duplicate each chain n GPU times



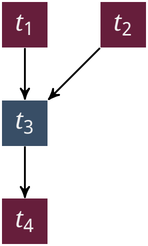
Graph expansion

Using all processing units

1. Breadth-first search to find **chains** of tasks that run on GPUs
2. Duplicate each chain n GPU times
3. Insert the new chain



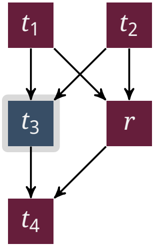
Cluster computing



Local master

Cluster computing

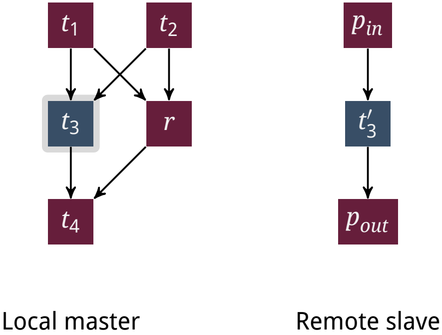
1. For each compute-intensive chain, insert a new remote node



Local master

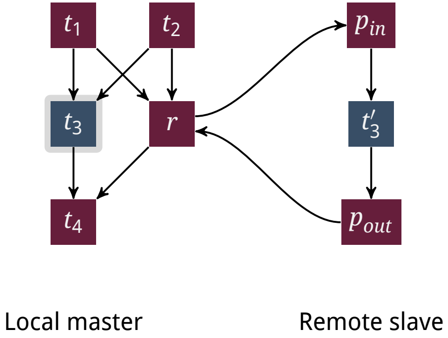
Cluster computing

1. For each compute-intensive chain, insert a new remote node
2. Send information to the remote and setup proxy nodes

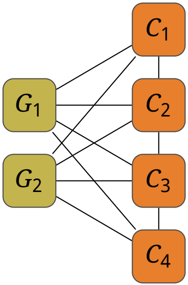


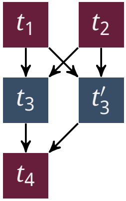
Cluster computing

1. For each compute-intensive chain, insert a new remote node
2. Send information to the remote and setup proxy nodes
3. Transfer data using ØMQ



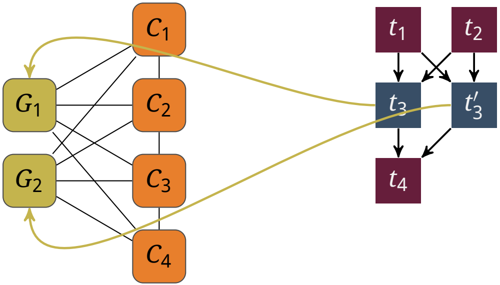
Heuristic, static scheduling





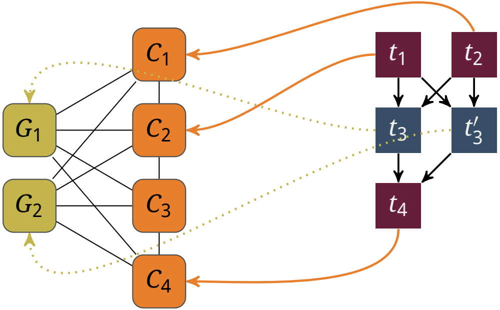
Heuristic, static scheduling

- Each chain of GPU nodes is mapped to one GPU



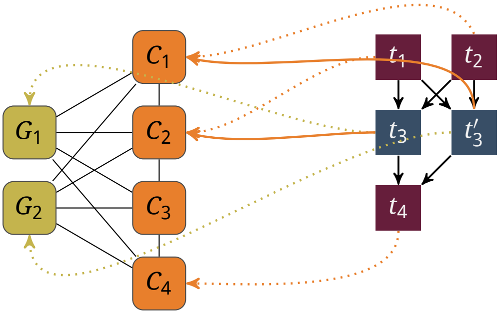
Heuristic, static scheduling

- Each chain of GPU nodes is mapped to one GPU
- Each CPU node is mapped to one of the CPUs



Heuristic, static scheduling

- Each chain of GPU nodes is mapped to one GPU
- Each CPU node is mapped to one of the CPUs
- A GPU node receives the same CPU affinity as the adjacent CPU node



Implementation Details

Dependencies & Tools

- Standard C99
- GLib/GObject + GObject introspection
- OpenCL 1.1 or 1.2
- ZeroMQ 3.2
- Nightly builds and unit test execution via Jenkins
- API documentation built with Gtk-Doc, manual with Sphinx

Implementation Details

Dependencies & Tools

- Standard C99
- GLib/GObject + GObject introspection
- OpenCL 1.1 or 1.2
- ZeroMQ 3.2
- Nightly builds and unit test execution via Jenkins
- API documentation built with Gtk-Doc, manual with Sphinx

High-level architecture

Implementation Details

Dependencies & Tools

- Standard C99
- GLib/GObject + GObject introspection
- OpenCL 1.1 or 1.2
- ZeroMQ 3.2
- Nightly builds and unit test execution via Jenkins
- API documentation built with Gtk-Doc, manual with Sphinx

High-level architecture

- Core framework manages OpenCL resources, graph and execution

Implementation Details

Dependencies & Tools

- Standard C99
- GLib/GObject + GObject introspection
- OpenCL 1.1 or 1.2
- ZeroMQ 3.2
- Nightly builds and unit test execution via Jenkins
- API documentation built with Gtk-Doc, manual with Sphinx

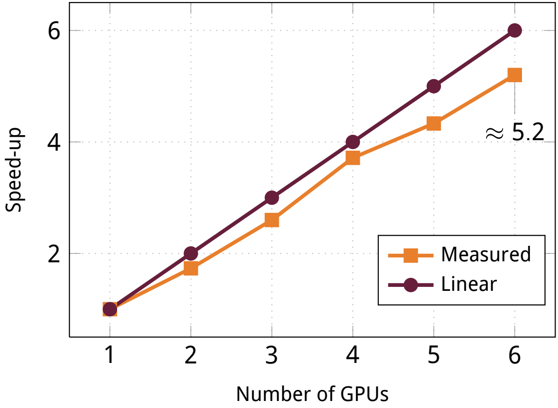
High-level architecture

- Core framework manages OpenCL resources, graph and execution
- Shared library plugins implement actual functionality (reading, writing, filtering, ...)

Scaling on a single compute node

Scaling on a single compute node

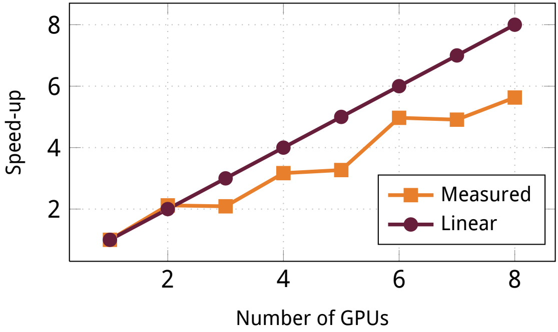
Tomographic reconstruction on a single compute server with 2 Xeon X5650 and 6 NVIDIA GTX 580.



Scaling on a (small) cluster

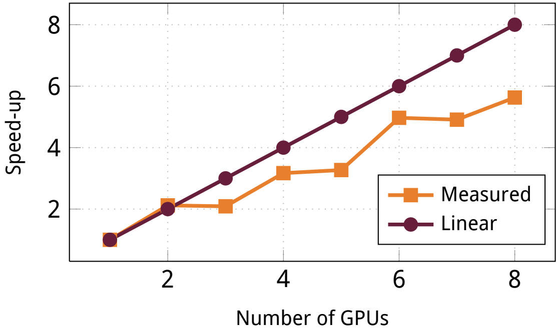
Scaling on a (small) cluster

4 GPU nodes connected via Infiniband, each node has 2 GTX 580.



Scaling on a (small) cluster

4 GPU nodes connected via Infiniband, each node has 2 GTX 580.



Some problems left ...

User interface

User interface

- Okay, none of you is interested in **how** things are performed

User interface

- Okay, none of you is interested in **how** things are performed
- Use as a C library

User interface

- Okay, none of you is interested in **how** things are performed
- Use as a C library
- Programmatic access through bindings (e.g. Python)

User interface

- Okay, none of you is interested in **how** things are performed
- Use as a C library
- Programmatic access through bindings (e.g. Python)
- Simplified Python wrapper for straightforward tasks

User interface

- Okay, none of you is interested in **how** things are performed
- Use as a C library
- Programmatic access through bindings (e.g. Python)
- Simplified Python wrapper for straightforward tasks
- JSON serialization format

User interface

- Okay, none of you is interested in **how** things are performed
- Use as a C library
- Programmatic access through bindings (e.g. Python)
- Simplified Python wrapper for straightforward tasks
- JSON serialization format
- WIP: Graphical user interface based on GTK+

Simple example using the bindings

```
from gi.repository import Ufo

pm = Ufo.PluginManager()
reader = pm.get_task('reader')
writer = pm.get_task('writer')
bp = g.get_task('backproject')

bp.props.axis_pos = 413.5
bp.props.rotation_angle = 0.012

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

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

Less code with the wrapper

```
import ufotools

factory = ufotools.Factory()
reader = factory.get('reader')
writer = factory.get('writer')
bp = factory.get('backproject',
                 axis_pos=413.5,
                 rotation_angle=0.012)

writer(bp(reader)).run()
```

No code with JSON

```
$ runjson description.json
```

Extending

Extending

Simple functionality

- Write OpenCL kernel code
- Use `opencl` task node and specify file and/or kernel name

Extending

Simple functionality

- Write OpenCL kernel code
- Use `opencl` task node and specify file and/or kernel name

More complex computing

- Create `.c` and `.h` files from a template script
- Implement method that receives a compute environment and data buffers to read from and write into

Extending

Simple functionality

- Write OpenCL kernel code
- Use `opencl` task node and specify file and/or kernel name

More complex computing

- Create `.c` and `.h` files from a template script
- Implement method that receives a compute environment and data buffers to read from and write into

Extending from within Python

- Numpy data sources and sinks possible
- Writing tasks **in** Python code is challenging due to the GIL

Cluster management

- No built-in management
- But could be added on a higher level based on existing primitives

```
[slaves]$ ufod --listen tcp://ib0:5555
[master]$ runjson description.json \
           -a tcp://192.168.60.10{2,3,4}:5555
```

Conclusion

UFO framework

Conclusion

UFO framework

- Provides a simple but powerful processing abstraction

Conclusion

UFO framework

- Provides a simple but powerful processing abstraction
- Utilizes multiple GPUs on multiple machines

Conclusion

UFO framework

- Provides a simple but powerful processing abstraction
- Utilizes multiple GPUs on multiple machines
- Is modular, extensible and **free** software

Conclusion

UFO framework

- Provides a simple but powerful processing abstraction
- Utilizes multiple GPUs on multiple machines
- Is modular, extensible and **free** software
- Can be integrated as a C library, via language bindings or a JSON description

Conclusion

Karlsruhe Institute of Technology

UFO framework

- Provides a simple but powerful processing abstraction
- Utilizes multiple GPUs on multiple machines
- Is modular, extensible and **free** software
- Can be integrated as a C library, via language bindings or a JSON description

More information at `ufo.kit.edu`

Thanks for your attention. Any questions?

Build procedure

```
$ git clone http://ufo.kit.edu/git/ufo-core
$ mkdir build
$ cd build
$ cmake ../ufo-core
$ make && make test && make install
```

or

```
$ sudo ufo-core/tools/deploy.sh $HOME/usr
```