Karlsruhe Institute of Technology

Real-time X-ray image reconstruction at ANKA

Matthias Vogelgesang
matthias.vogelgesang@kit.edu

Institute for Data Processing and Electronics

X-ray imaging

Karlsruhe Institute of Technology

- Non-destructive imaging technique

X-ray imaging

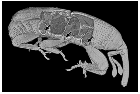
- Non-destructive imaging technique in two,

Radiograph

X-ray imaging

- Non-destructive imaging technique in two, three

Radiograph

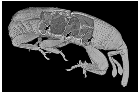


Tomogram

X-ray imaging

- Non-destructive imaging technique in two, three and four dimensions

Radiograph



Tomogram

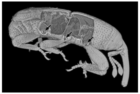
Sequence of tomograms

X-ray imaging

Karlsruhe Institute of Technology

- Non-destructive imaging technique in two, three and four dimensions

Radiograph

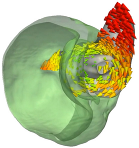


Tomogram

Sequence of tomograms

Examples

- Morphology studies
- Cement hardening and cracking behaviour
- Dispersion of oil after leaving micro valves
- Bubble forming
- …



Hip-join screw in *Trigonopterus* (v.d.Kamp)

Ångströmquelle Karlsruhe (ANKA)

Karlsruhe Institute of Technology

Synchrotron radiation

- Accelerated electrons emit X-ray (aka synchrotron) radiation
- X-ray source produces beam with broad spectrum and high intensity
- X-ray scattering and diffraction, high-resolution and high-speed radiography, tomography and laminography

image beamline currently commissioned for fast micro tomography.

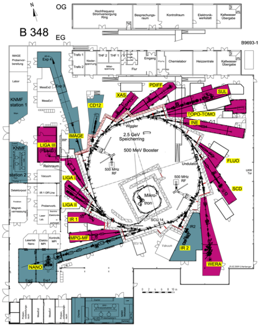


Figure : Floor plan of anka electron ring with 16 tangential beamlines

Basic tomography experiment

Beamline

X-Ray → Filter → Monochr.

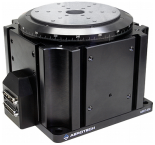
Basic tomography experiment

Karlsruhe Institute of Technology

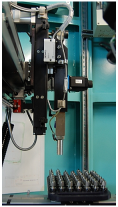
Beamline

Sample stage

X-Ray → Filter → Monochr. → Sample → Scint. → Optics → Detector



Rotary stage



Sample changer

Basic tomography experiment

Beamline: X-Ray → Filter → Monochr.

Sample stage: Sample → Scint. → Optics → Detector

100 frames/s @ 16 Bit

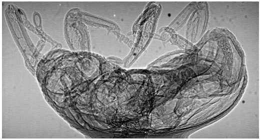
100000 frames/s @ 12 Bit

Basic tomography experiment

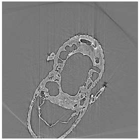
Beamline

Sample stage

X-Ray → Filter → Monochr. → Sample → Scint. → Optics → Detector → Process.



Projections



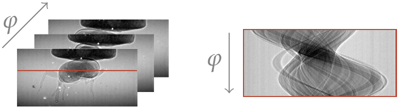
Reconstructed slice

Visualization

Tomographic reconstruction

Problem

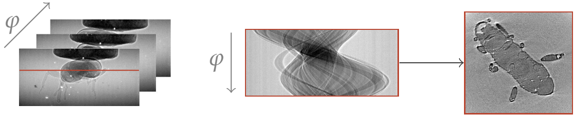
From a series of projections …



Tomographic reconstruction

Problem

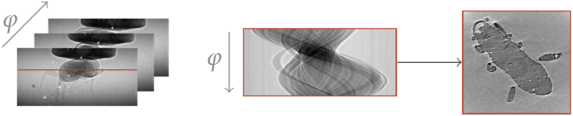
From a series of projections … reconstruct *unknown* slice information



Tomographic reconstruction

Problem

From a series of projections … reconstruct *unknown* slice information



Solutions

- Analytically using Fourier-slice theorem (dfi)
- Filter projections and smear back into empty volume (fbp)
- Model detection as a linear system and solve algebraically (art)

Trade-off in terms of quality and performance but in any case too slow on CPUs

Challenges

Large data volumes

- Up to hundred of samples per experiment scan
- 1000 projections per sample taken in < 100 ms
- Per projection up to eight mega pixels at 16 bits

8 GB raw input $\rightarrow$ 128 GB output, per scan!

Challenges

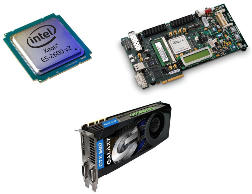
Large data volumes

- Up to hundred of samples per experiment scan
- 1000 projections per sample taken in < 100 ms
- Per projection up to eight mega pixels at 16 bits

8 GB raw input $\rightarrow$ 128 GB output, per scan!

Efficient data processing

- Compute-intensive algorithms on data streams
- Must use existing infrastructure, heterogeneous systems as well as small-scale clusters
- Easy to use for physicists and visiting researchers



How do we tackle the challenges?

1. Use GPU implementations of the algorithms for highest possible FLOPs
2. Use a framework that executes the algorithms on heterogeneous systems
3. Provide user-friendly high-level abstractions

GPU image processing

Image processing on the GPU

Karlsruhe Institute of Technology

General approach

- Process each output pixel/voxel in one thread
- Optimize for each hardware generation and vendor
- Use optimized libraries for certain problems

Optimizations

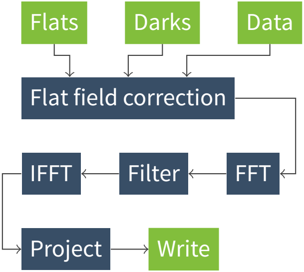
- Texture lookups for random access
- Local memory storage for repeated data access
- Reduction of kernel size and register usage
- Work group size adjustments

An OpenCL framework

Basic idea

Graph-based task descriptions

- Define algorithmic computation and data flow as a directed graph of atomic tasks
- Lends itself to task and pipeline parallelism
- “Somehow” schedule work on available resources



Initial approaches

Karlsruhe Institute of Technology

Local scheduling

- Run each task in its own thread and use a currently unused device from a pool
- Works well on single GPU machines
- …but fails miserably on multi-GPU machine due to excessive data transfers

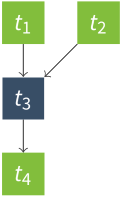
Group scheduling

- Extend local scheduling by duplicating tasks per node
- Same performance in single GPU case
- Better for multiple GPUs, but does not scale sufficiently

Current approach

Static scheduling

- The data pipeline is static, so can be the schedule

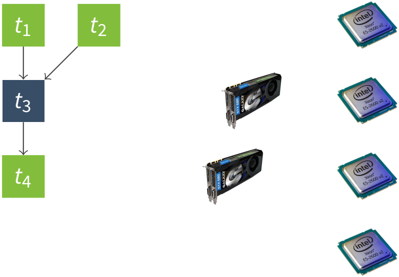


Current approach

KIT Karlsruhe Institute of Technology

Static scheduling

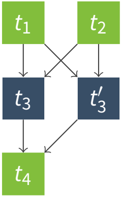
- The data pipeline is static, so can be the schedule
- Determine processing resources

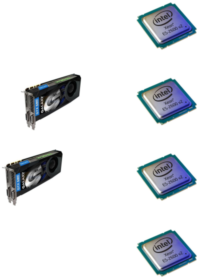


Current approach

Static scheduling

- The data pipeline is static, so can be the schedule
- Determine processing resources
- Transform graph to accomodate for additional processors

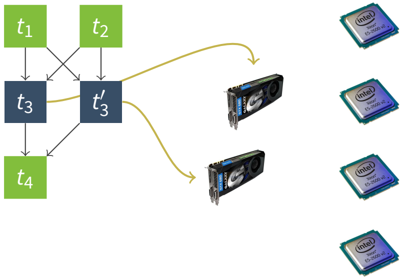




Current approach

Static scheduling

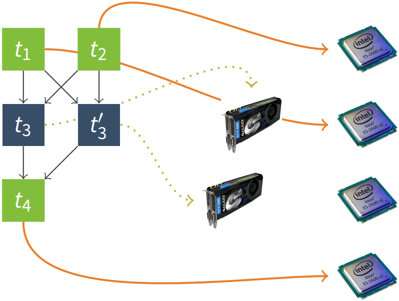
- The data pipeline is static, so can be the schedule
- Determine processing resources
- Transform graph to accomodate for additional processors
- Assign tasks to GPUs



Current approach

Static scheduling

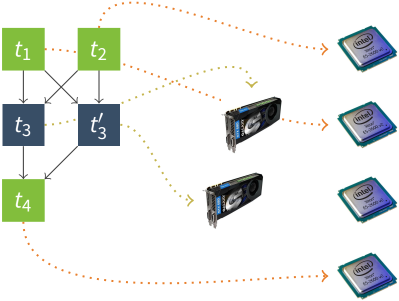
- The data pipeline is static, so can be the schedule
- Determine processing resources
- Transform graph to accomodate for additional processors
- Assign tasks to GPUs and CPUs



Current approach

Static scheduling

- The data pipeline is static, so can be the schedule
- Determine processing resources
- Transform graph to accomodate for additional processors
- Assign tasks to GPUs and CPUs
- Execute with fixed assignment



Analysis

Pros

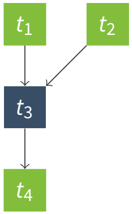
- Sequences of tasks are replicated for additional GPUs
  - Adjacent tasks share data with double buffer scheme
- Excessive data transfers are reduced by keeping data local
- Relatively easy to implement
- Is easily extended to other resources

Cons

- Double buffering is counter-productive when neighbouring tasks *have to* share data

Scaling to clusters

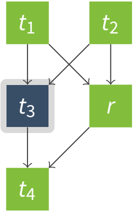
- Use same algorithmic description



Local master

Scaling to clusters

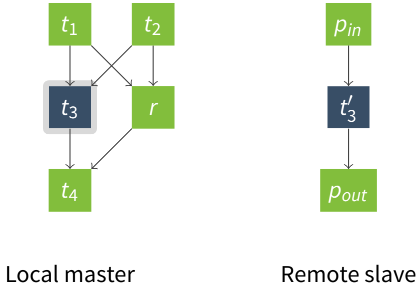
- Use same algorithmic description
- Add proxy tasks for sub-graph



Local master

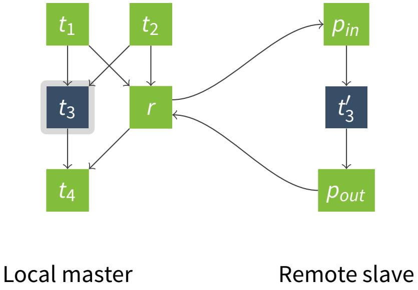
Scaling to clusters

- Use same algorithmic description
- Add proxy tasks for sub-graph
- Instatiate tasks on remote nodes



Scaling to clusters

- Use same algorithmic description
- Add proxy tasks for sub-graph
- Instatiate tasks on remote nodes
- Forward data and receive results

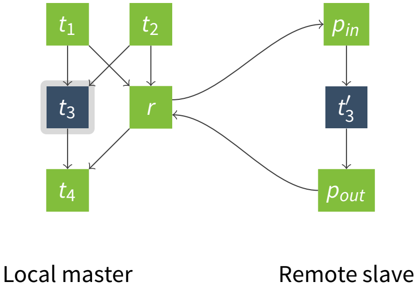


Scaling to clusters

- Use same algorithmic description
- Add proxy tasks for sub-graph
- Instatiate tasks on remote nodes
- Forward data and receive results

Open questions

- One proxy task per remote resource or a single one?
- Remote execution has impact on scheduling



Usage

Using the system

- System is a set of C libraries and shared object plugins built on top of GObject and OpenCL
- Simple programmatic access via API or JSON serialization

```
fft = plugin_manager_get_task ("fft");
/* ... */
set_property(filter, "type", "butterworth");
/* ... */
graph_connect_nodes (graph, fft, filter);
/* ... */
scheduler_run (graph)
```

- Not very appealing for quick prototyping

Using through high-level abstraction

Python integration

- GObject gives us bindings for free, we just provide a thin convenience layer
- Implicit data conversion and scheduler invocation
- Push and pull data in generator fashion

Example

```python
from ufo import Backproject, Reader

reader = Reader(filename='foo-%05i.tf')
bp = Backproject()

for result in bp(reader()):
    pyplot.imshow(result)
```

Extending the system

On a lower level

- Implement in C, flawless but quite some work
- Implement in Python, works …unreliably
- Write kernel code and use provided OpenCL filter

On a higher level

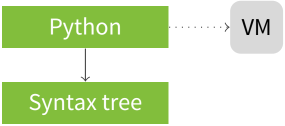
Write Python code and *generate* OpenCL kernel

Compiling Python to OpenCL

- Normally, Python interprets statements on its own virtual CPU

Compiling Python to OpenCL

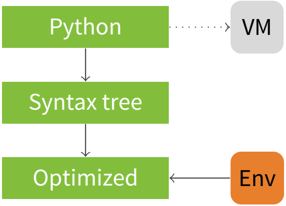
- Normally, Python interprets statements on its own virtual CPU
- Instead, let's parse code into syntax tree



Compiling Python to OpenCL

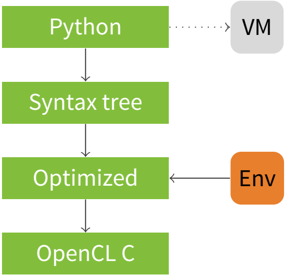
Karlsruhe Institute of Technology

- Normally, Python interprets statements on its own virtual CPU
- Instead, let’s parse code into syntax tree
- Optimize tree w.r.t. to resources



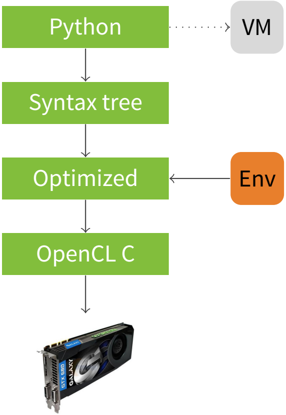
Compiling Python to OpenCL

- Normally, Python interprets statements on its own virtual CPU
- Instead, let's parse code into syntax tree
- Optimize tree w.r.t. to resources
- Generate OpenCL C (maybe SPIR soon)



Compiling Python to OpenCL

- Normally, Python interprets statements on its own virtual CPU
- Instead, let's parse code into syntax tree
- Optimize tree w.r.t. to resources
- Generate OpenCL C (maybe SPIR soon)
- Pass code to our framework or run with PyOpenCL



Optimization opportunities

- Static analysis
  - Pre-compute values
  - Simplify expressions
  - Replace expressions with native OpenCL constructs
- Dynamic analysis
  - Determine alternative address spaces by analysing access pattern and data size
  - Pre-compute sizes and bounds, i.e. using `len()` on data

Using the translator

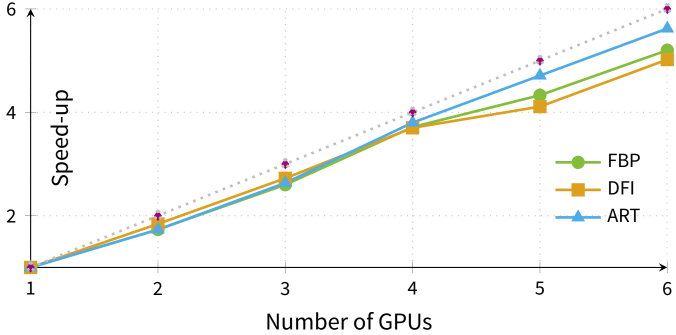
- Use decorators to compile at import or run time

```
@jit
def compute(x, y):
    return np.cos(x) + np.sin(y)
```

- Calling `compute` either returns or runs the kernel
- Python constructs are mapped to suit image processing
  - Relative indexing via `x[+i]` and `x[-i]`
  - Unindexed access computes on all elements
  - Running `for` over data iterates over elements
  - ...

Results

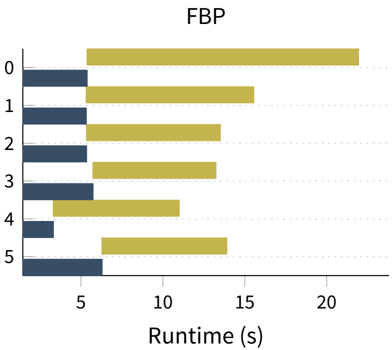
Single node multi-GPU



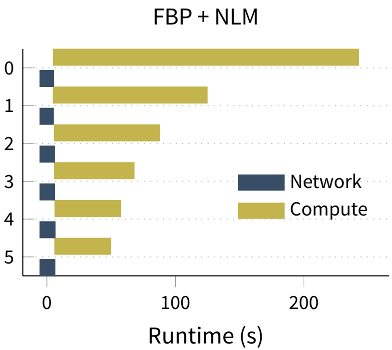
Good scalability with near linear speed-up for up to 6 NVIDIA GTX 580's.

Multi node cluster

Scalability strongly depends on “computation / data transfer” ratio . . .

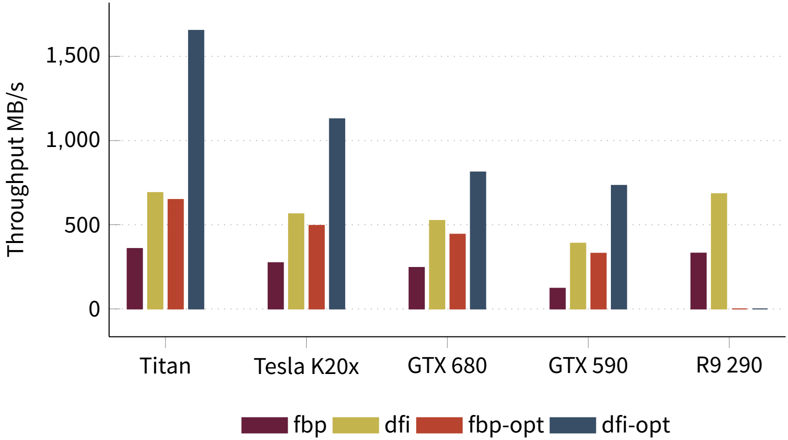


High network overhead

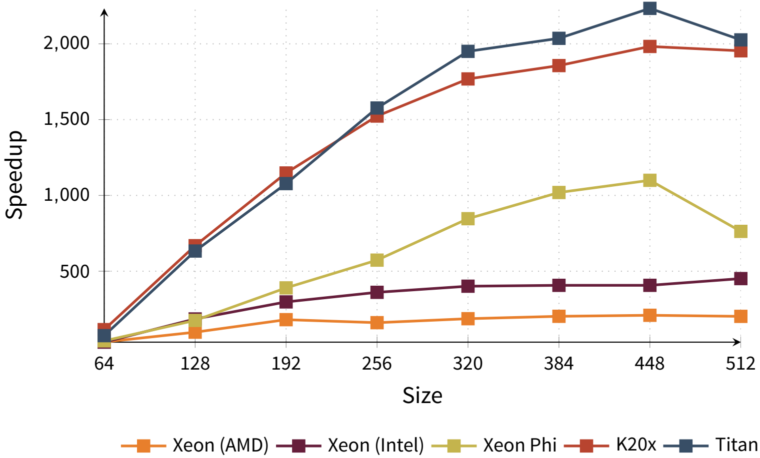


Almost linear speedup

Real-time capable?



Aside: generated kernel vs. NumPy



Conclusion

Karlsruhe Institute of Technology

Where are we?

- We can reconstruct in "real-time" and assess quality ✓
- We scale with multiple GPUs and compute nodes ✓
- We can prototype GPU algorithms ✓

Conclusion

Karlsruhe Institute of Technology

Where are we?

- We can reconstruct in “real-time” and assess quality ✓
- We scale with multiple GPUs and compute nodes ✓
- We can prototype GPU algorithms ✓

Open questions, future work

- Should we use a DSL (e.g. Julia, Halide, …) and use our system as a “backend?”
- Kernel fusion and fission
- What about SPIR instead of OpenCL C?

Acknowledgements

- Institute for Photon Science and Synchrotron Radiation (ips)
- Helmholtz Zentrum Geesthach (hzg) and Deutsches Elektronen-Synchrotron (desy)
- U Tomsk, U Moscow and St. Petersburg

Thank you. Any questions?

`github.com/ufo-kit`