STIR and Tensorflow

Philipp Windischhofer

July 27, 2017

The big picture

- STIR
- iterative reconstruction algorithms
- find the optimum of a *cost function*

$$P(\text{global LOR response}|\text{image}) = \prod_{\text{all LORs}} P(\text{LOR response}|\text{image})$$

$$P(\text{LOR response}|\text{image}) = \int_{\text{LOR}} \text{image density}$$

- Image (estimate) $\rightsquigarrow$ LOR response: *forward projection*

Ray tracing

- in the following: how to compute / $\underline{\text{approximate}}$ this integral?
- represent *image* as a discretized array of voxels

$$\int_{\text{LOR}} \text{image density} \rightsquigarrow \sum_{\text{traversed voxels}} \text{LOI} \cdot \text{voxel brightness}$$

- LOI $=$ $\underline{\text{length of intersection}}$ of the LOR through a specific voxel

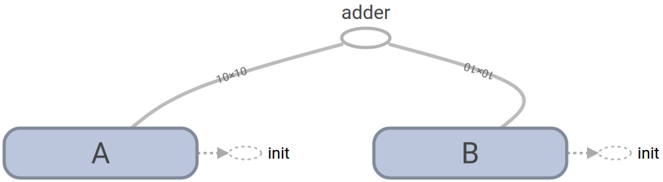
(Standard) STIR spends $> 90\%$ of the reconstruction time for *forward projection*

- is there a way to make it faster?

Table of Contents

Tensorflow

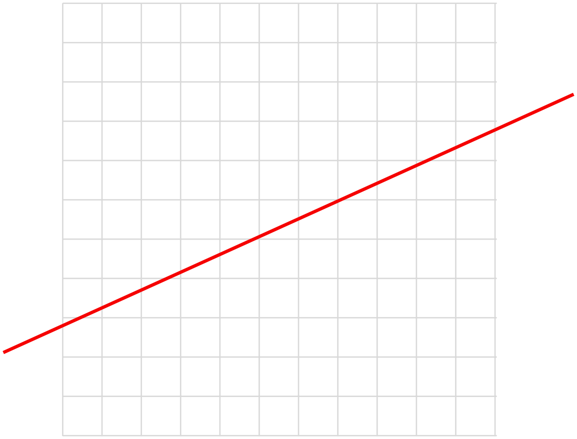
- *Google*: “An open-source software library for machine intelligence”
  - heavily used in the machine-learning community
- Much more than that:
  - a software package for numerical calculations using data-flow *graphs*



  - Design the graph *now*, save it to a file, run it *later*...
  - Can run it (almost) everywhere: CPU, GPU, a cluster, ...

How to do ray tracing on a CPU?

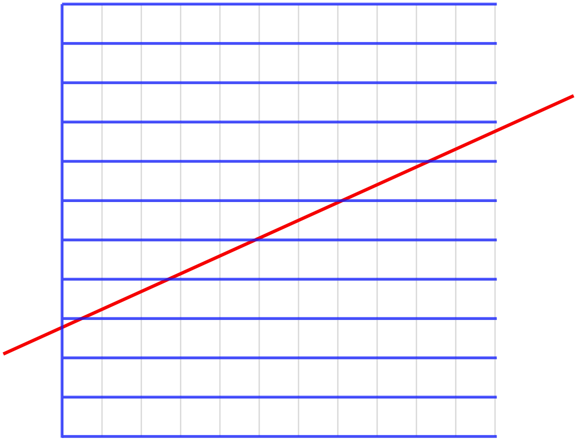
Siddon's algorithm (2D)



- a (very) sequential algorithm

How to do ray tracing on a CPU?

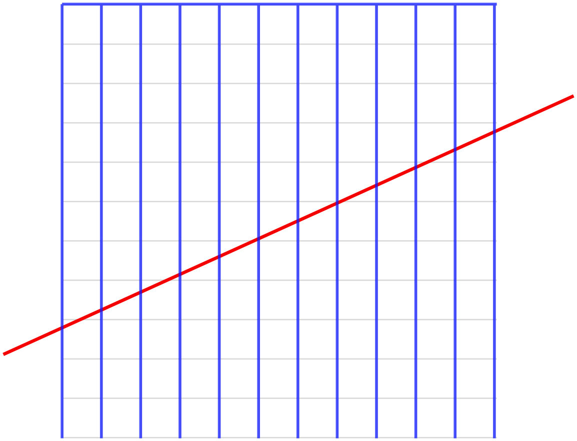
Siddon's algorithm (2D)



- get intersections of LOI with all horizontal planes

How to do ray tracing on a CPU?

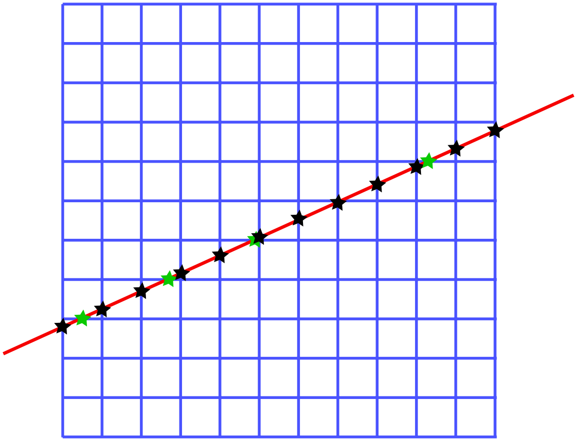
Siddon's algorithm (2D)



- get intersections of LOI with all vertical planes

How to do ray tracing on a CPU?

Siddon's algorithm (2D)



- combine them and get the individual LOIs

How to do ray tracing on a CPU?

Siddon's algorithm (2D)

- parametrize the LOR

$$\mathbf{x} = (\mathbf{x}_{\mathbf{end}} - \mathbf{x}_{\mathbf{start}}) \cdot \alpha + \mathbf{x}_{\mathbf{start}}$$

$$\alpha \in [0, 1]$$

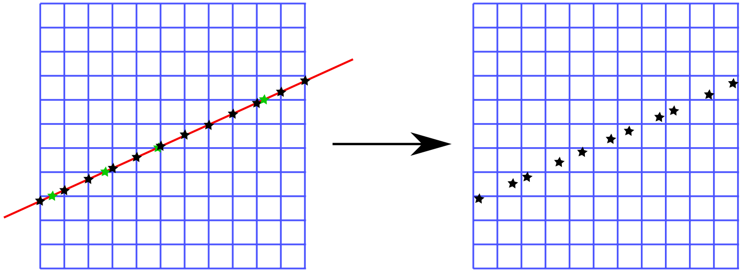
- get intersections with horizontal and vertical planes in terms of the line parameter α
- merge and sort them
- *differences* between neighbouring α values in the merged list describe the LOIs!

Siddon and Tensorflow

- Siddon: $\mathcal{O}(N)$ for a LOR of length N (this means a voxel array of size N^3!!)
- many branches, a loop over the LOR, ...
- very inefficient when running with Tensorflow (GPU $\neq$ CPU architecture)
  - especially problematic when trying to trace multiple LORs "in parallel"
- for Tensorflow: need an algorithm that ...
  - uses the same algorithmic steps on *all* voxels of the LOR
  - uses the same algorithmic steps on *all* LORs

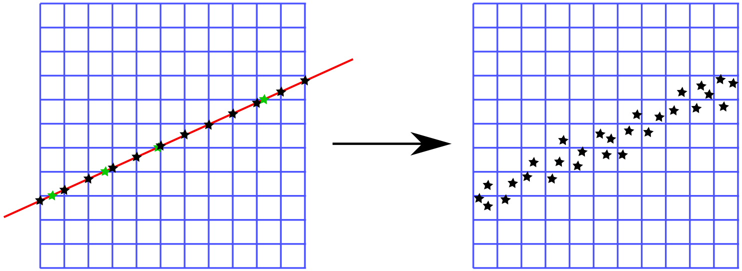
Alternatives to Siddon

- naive approach: for every LOR, compute the LOI through *every* voxel, then add them all up
  - works fine with Tensorflow, but scales as $\mathcal{O}(N^3)$
  - CPU & Siddon beat it again for real-world array sizes
- alternative approach: think of (intersection) points *without* the notion of an LOR



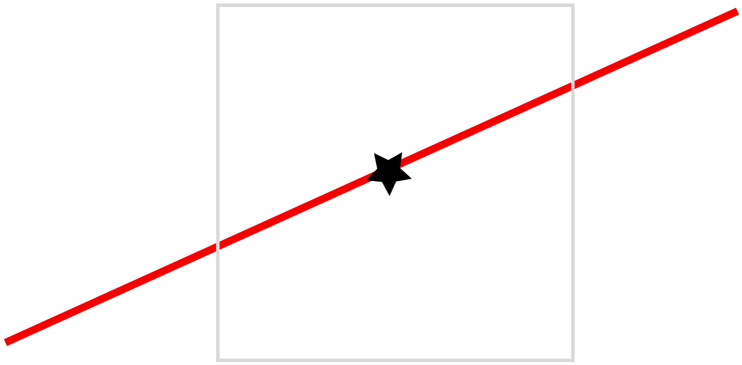
Alternatives to Siddon

- naive approach: for every LOR, compute the LOI through *every* voxel, then add them all up
  - works fine with Tensorflow, but scales as $\mathcal{O}(N^3)$
  - CPU & Siddon beat it again for real-world array sizes
- alternative approach: think of (intersection) points *without* the notion of an LOR



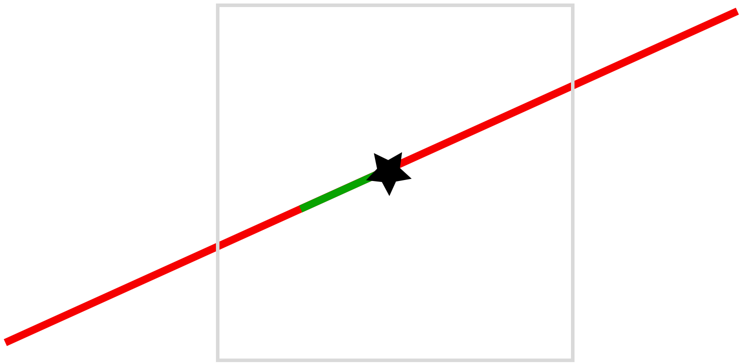
Voxel-by-voxel LOI calculation

- still not much easier than the full problem!
- a cube is very discontinuous, hard to do it analytically without branches
- what if content with an *approximate* solution?



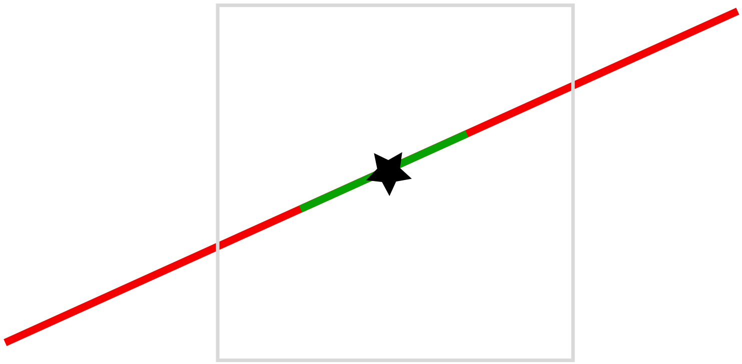
Voxel-by-voxel LOI calculation

- still not much easier than the full problem!
- a cube is very discontinuous, hard to do it analytically without branches
- what if content with an *approximate* solution?



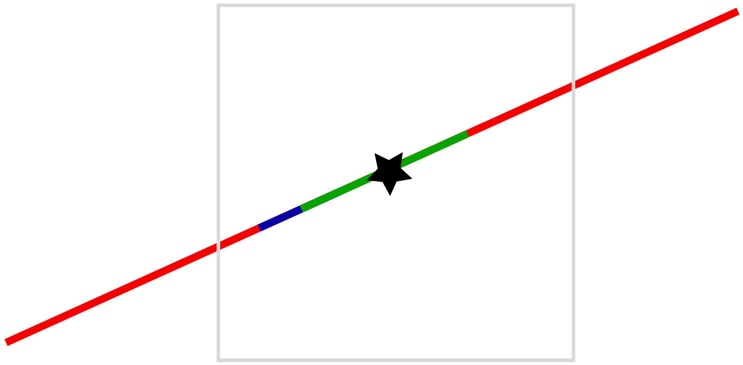
Voxel-by-voxel LOI calculation

- still not much easier than the full problem!
- a cube is very discontinuous, hard to do it analytically without branches
- what if content with an *approximate* solution?



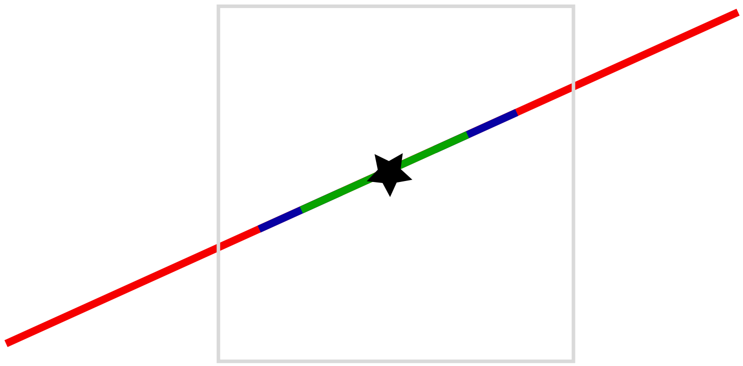
Voxel-by-voxel LOI calculation

- still not much easier than the full problem!
- a cube is very discontinuous, hard to do it analytically without branches
- what if content with an *approximate* solution?



Voxel-by-voxel LOI calculation

- still not much easier than the full problem!
- a cube is very discontinuous, hard to do it analytically without branches
- what if content with an *approximate* solution?



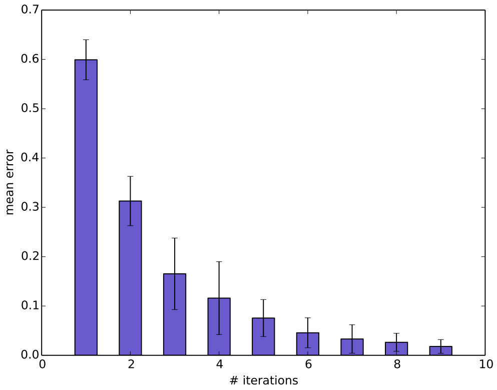
Ray marching: an iterative algorithm

- Use *signed distance function* of a cube to cut down the number of iterations that are needed
  - a heuristic that produces an *underestimation* of the distance to the voxel boundary
- need to assess the residual error that remains

Ray marching: an iterative algorithm

Accuracy

- assume: have “enough” points along the LOR



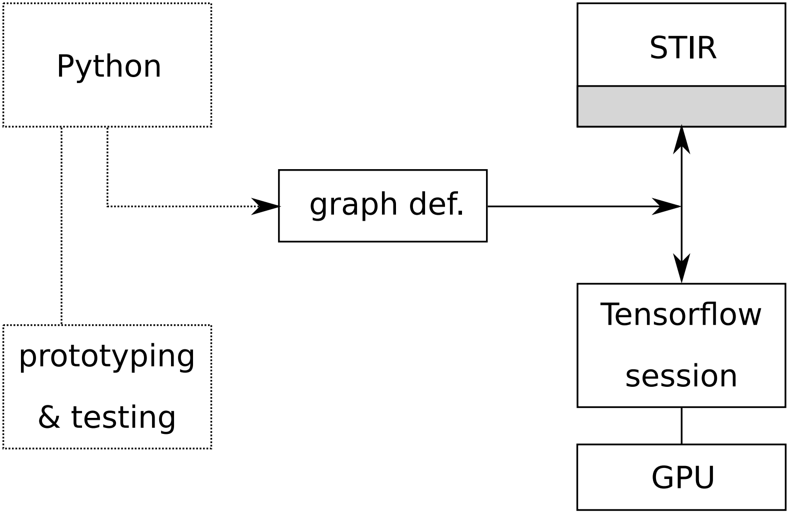
- with 6 iterations, are already at $\sim 5\%$ level!

Bringing together STIR and Tensorflow

The infrastructure

- Build the Tensorflow graph in a Python script
  - much easier and faster to prototype
  - Python API much better documented and more stable
- <u>Save</u> the complete graph into a `ProtoBuf` file
- <u>Load</u> the file using the C++ API and make use of it in STIR
  - link STIR against the Tensorflow (shared) library

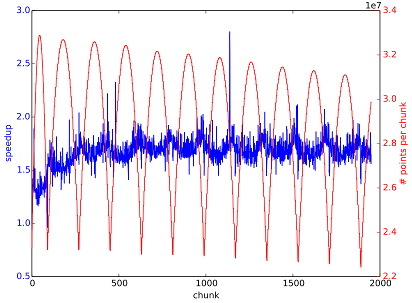
The workflow

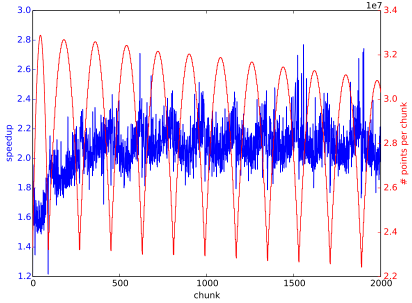


Caching in STIR

- scanners have (many) symmetries
- `ProjMatrixByBin` maintains a cache
  - already computed matrix elements are stored
  - new matrix elements related by symmetries to known ones "come for free"!
- but: caching most efficient if *every* new matrix element request is checked against cache before computing
- *batching* the matrix element requests makes caching very inefficient!

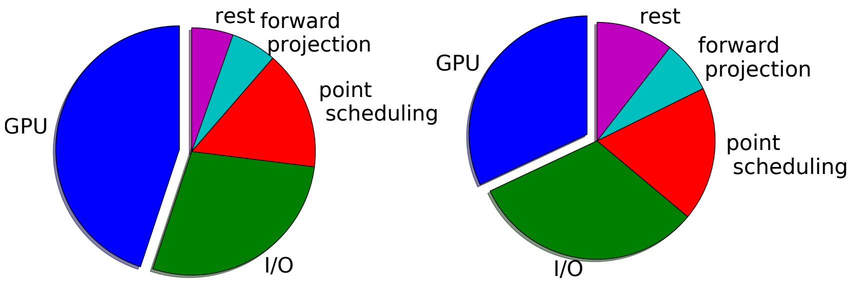
Speedup without caching





left: 6 iterations, **right**: 2 iterations

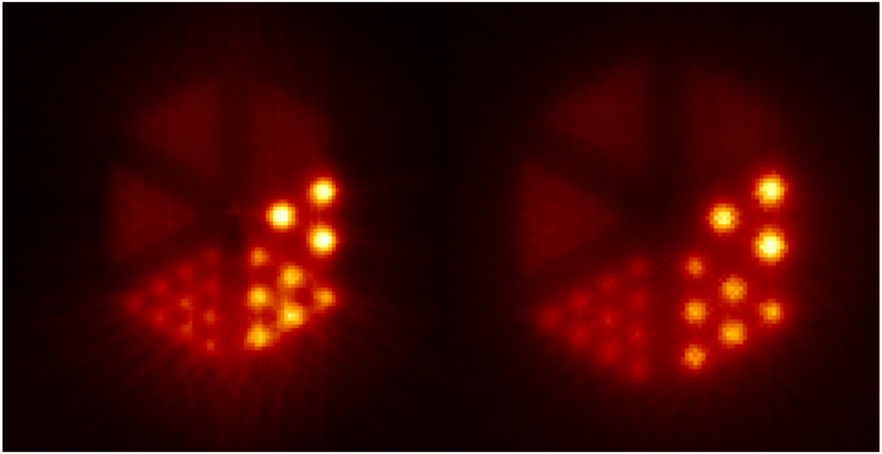
- average speedup is similar



left: 6 iterations, **right**: 2 iterations

- *I/O*: converting from `ProjMatrixElemsForOneBin` to `Tensor` and back
- *point scheduling*: choose points to sample the TOR / LOR

Images



left: 2 iterations, 20LORs per matrix element, **right**: original STIR

How to proceed?

- demonstrated a proof-of-concept of integrating Tensorflow into STIR
- whole toolchain is in place
  - separation of graph generation (Python) and usage (C++) works very well
- Speedup of ~ 2 observed

Not too bad, but still some limitations:

- how to improve utilization of GPU (conversion between Tensors and STIR objects)?
- is there a better way to estimate the matrix elements?
- so far, use only a single thread (no `openMP` support)
  - Tensorflow sessions tend to allocate all of the available GPU memory to avoid fragmentation...

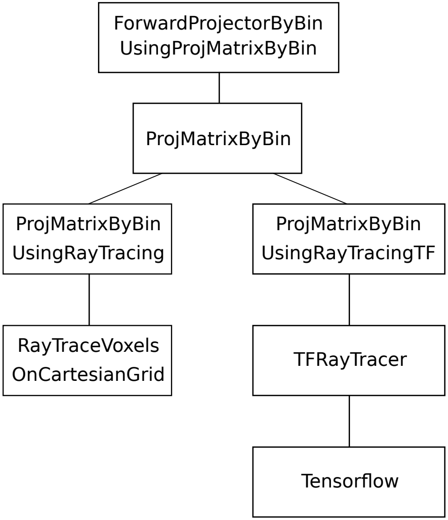
Where to find the code?

- STIR-TF: `https://github.com/philippwindischhofer/STIR/tree/stir-tf`
  - in particular: `https://github.com/philippwindischhofer/STIR/blob/stir-tf/documentation/statistics/doc/main.pdf`
- ray tracing scripts: `https://gitlab.phys.ethz.ch/luster/tf-raytracing` (ask Werner for access rights)

Any comments and contributions are welcome!

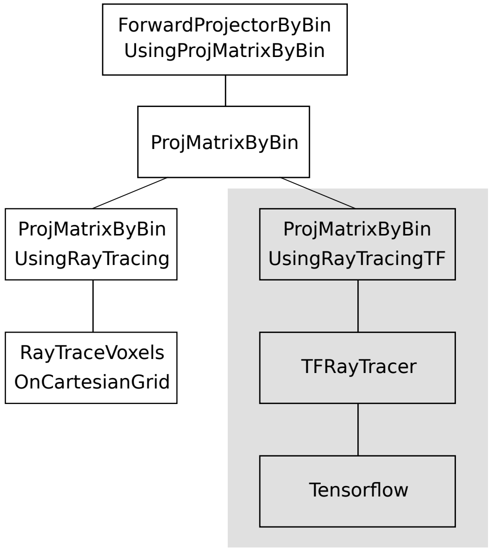
Backup

Tensorflow in STIR



ForwardProjectorByBin UsingProjMatrixByBin
ProjMatrixByBin
ProjMatrixByBin UsingRayTracing
ProjMatrixByBin UsingRayTracingTF
RayTraceVoxels OnCartesianGrid
TFRayTracer
Tensorflow

Tensorflow in STIR


ForwardProjectorByBin UsingProjMatrixByBin
ProjMatrixByBin
ProjMatrixByBin UsingRayTracing
ProjMatrixByBin UsingRayTracingTF
RayTraceVoxels OnCartesianGrid
TFRayTracer
Tensorflow