Design of a Modular CT Reconstruction Framework

Max Aehle

Chair for Scientific Computing,
TU Kaiserslautern, Germany
& SIVERT Research Training Group

July 18^{th}, 2022

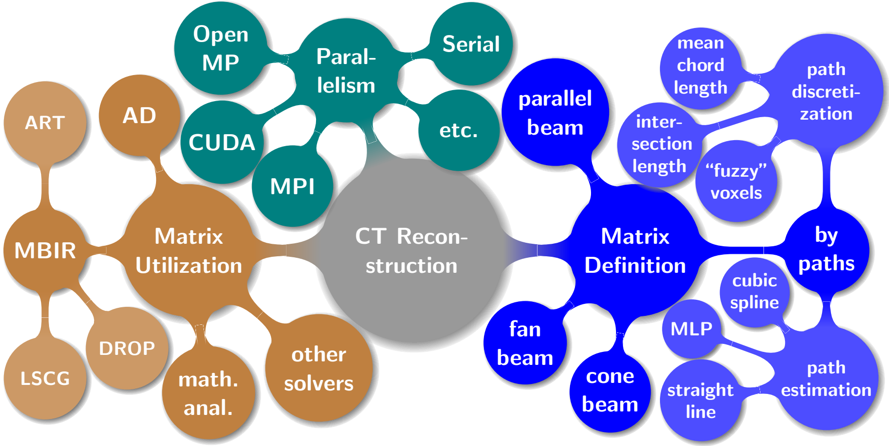
Outline

1. Problem: Decouple matrix generation, matrix utilization, and parallelism.
2. Solution: Policy pattern.
3. Some intermediate results.

CT Reconstruction ...

- ... basically solves a linear system $A \cdot x = b$ approximately.
- In list-mode proton CT, A_{ij} relates the RSP x_j of voxel j and the WEPL b_i of proton i.
- In X-ray CT, A_{ij} relates the radiodensity x_j of voxel j and the attenuation b_i of ray i.
- A is big but sparse.
- Use matrix-free solvers which e. g. perform $(A\cdot)$, $(A^T\cdot)$ or compute $(\|A_{i,\cdot}\|_2)_i$.
- These tasks can be easily parallelized.
- Designing an algorithm involves a lot of choices!

... many "orthogonal" choices.

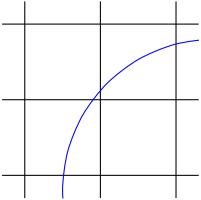


Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.

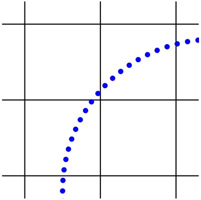
Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.



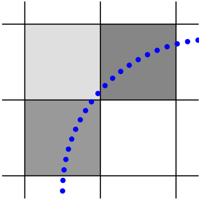
Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.
- Path discretization:



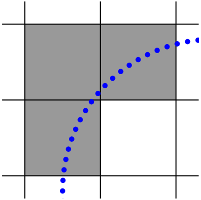
Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.
- Path discretization:
  - approximate intersection lengths



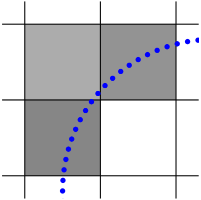
Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.
- Path discretization:
  - approximate intersection lengths
  - fixed entry for all traversed voxels



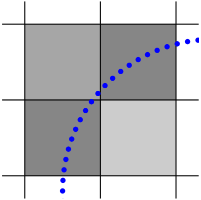
Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.
- Path discretization:
  - approximate intersection lengths
  - fixed entry for all traversed voxels
  - mean chord length

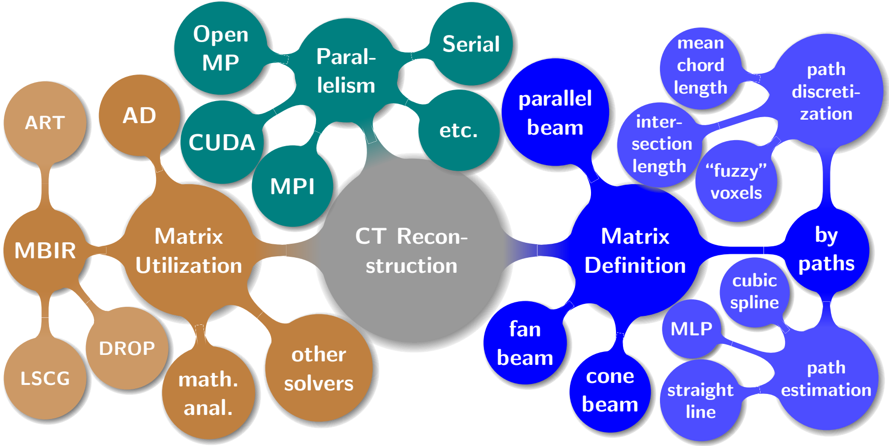


Matrix Definition

- The matrix element A_{ij} is related to the length of intersection between voxel j and "ray"/path i.
- In X-ray CT, we might generate matrix row-wisely (find voxels on ray, for each ray) and column-wisely (find rays through voxel, for each voxel).
- In particle CT, only the row-wise approach is efficient.
- Path estimation: linear path, cubic spline, (extended) most likely path.
- Path discretization:
  - approximate intersection lengths
  - fixed entry for all traversed voxels
  - mean chord length
  - "fuzzy voxels"



. . . many “orthogonal” choices.



Make it Modular!

To avoid code duplication, split matrix definition and matrix utilization in the code. Possible interfaces:

- *Storing* the matrix would be straightforward, but is infeasible for big problems as memory is limited.
- *Policy pattern* $\rightsquigarrow$ next slides.

The logic to e. g. copy data to GPU and launch kernels should also be decoupled from the matrix definition.

Policy pattern

A (row-ordered) sparse matrix class `M` must define `rows()`, `cols()`, and

```
template<typename fun_t>
__host__ __device__
void assembleRow(int row, fun_t fun) const {
  // compute elements of row and for each
  // of them, call
  fun(col, val);
}
```

`M` inherits from `SparseRowOrdered<M>` which defines functions like `multiply_serial`, `multiply_openmp`, `multiply_cuda`, and a dispatcher `multiply`.

Multiplication in Serial

```
bool multiply_serial(DataVector<const Scalar> const& x,
                     DataVector<Scalar> const& y) const final {

  for(int row=0; row<rows(); row++){
    Scalar result = 0.;
    child()->assembleRow(row,
      [x,&result] (int col,Scalar val)->void {
        result += x[col] * val;
      }
    );
    y[row] = result;
  }
  return true;
}
```

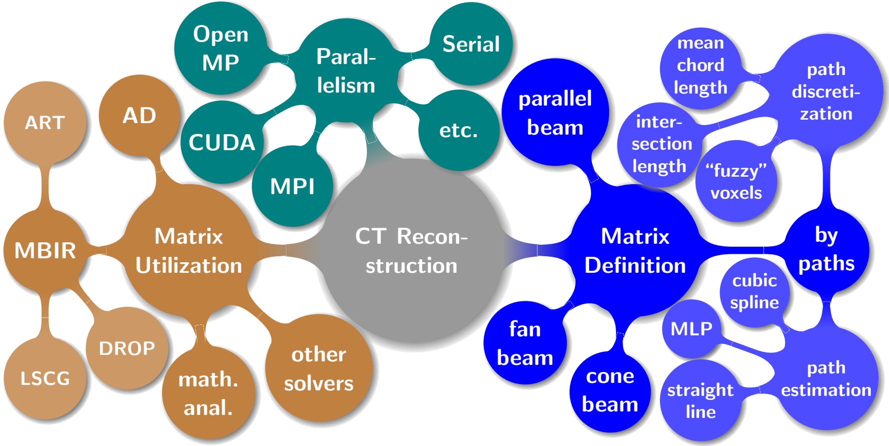
Multiplication with OpenMP

```
bool multiply_openmp(DataVector<const Scalar> const& x,
                     DataVector<Scalar> const& y) const final {
  #pragma omp parallel for
  for(int row=0; row<rows(); row++){
    Scalar result = 0.;
    child()->assembleRow(row,
      [x,&result] (int col,Scalar val)->void {
        result += x[col] * val;
      }
    );
    y[row] = result;
  }
  return true;
}
```

Multiplication with CUDA

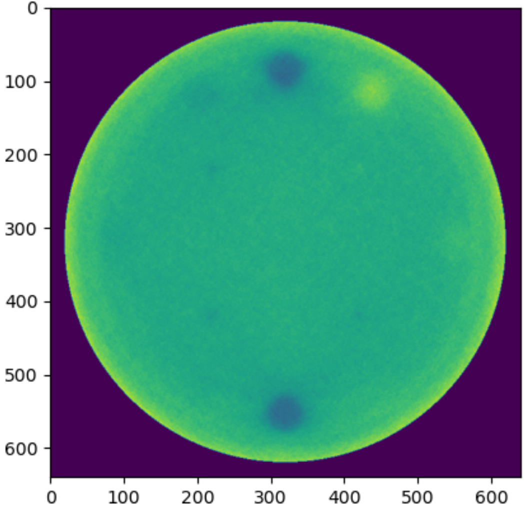
```
bool multiply_cuda(DataVector<const Scalar> const& x,
                   DataVector<Scalar> const& y) const final {
  // copy *this (i.e. the matrix itself) to the device
  Child* this_on_device = ...;
  // copy x to the device ...
  multiply_CudakernelRow<ThisClass>
    <<<dimGrid(rows()),dimBlock(rows())>>>
    (this_on_device, x.pdevice(), y.pdevice(), rows());
  // copy y from the device ...
  return true;
}
template <typename SparseRowOrdered_t>
__global__ void multiply_CudakernelRow(SparseRowOrdered_t const* A,
  double const* x, double* y, int nrows){
  unsigned int row = blockIdx.x*blockDim.x + threadIdx.x;
  if(row<nrows) {
    double result = 0.;
    A->child()->assembleRow(row,
      [x,&result] __device__ (int col,double val)->void {
        result += x[col] * val;
      }
    );
    y[row] = result;
  }
}
```

. . . many "orthogonal" choices.



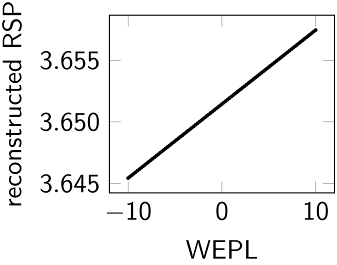
Intermediate Results

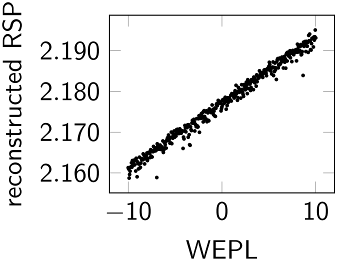
CTP 404, reconstructed from about 60e6 protons using DROP, $640 \times 20 \times 640$, several hours on single NVIDIA A100.



Intermediate Results

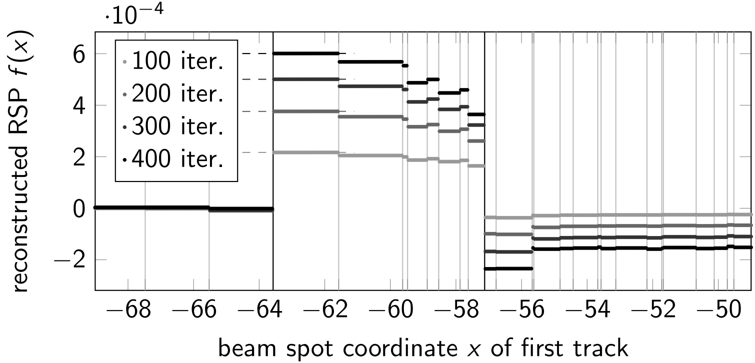
Black-box differentiation of DROP (left) should probably work, but LSCG (right) adds noise.





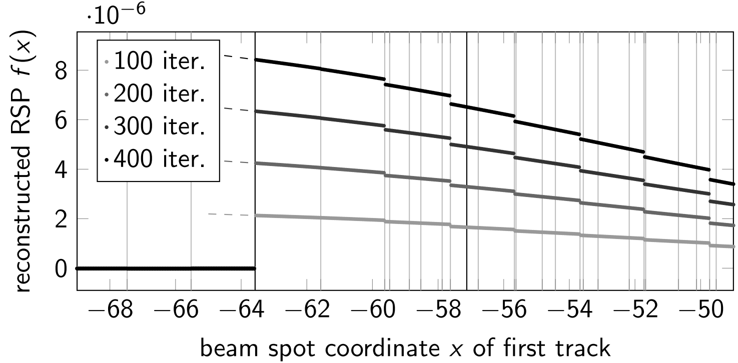
Intermediate Results

Derivatives w. r. t. proton track position coordinates do not carry much information, if a mean chord length approach is used:



Intermediate Results

When using a "fuzzy voxels" approach, there are still jumps but the derivatives now reflect the function's behaviour in a wider range:



The Bergen pCT Collaboration and SIVERT Research Training Group

- University of Bergen, Norway
- Helse Bergen, Norway
- Western Norway University of Applied Science, Bergen, Norway
- Wigner Research Center for Physics, Budapest, Hungary
- DKFZ, Heidelberg, Germany
- Saint Petersburg State University, Saint Petersburg, Russia
- Utrecht University, Netherlands
- RPE LTU, Kharkiv, Ukraine
- Suranaree University of Technology, Nakhon Ratchasima, Thailand
- China Three Gorges University, Yichang, China
- University of Applied Sciences Worms, Germany
- University of Oslo, Norway
- Eötvös Loránd University, Budapest, Hungary
- TU Kaiserslautern, Germany

Max Aehle:
max.aehle@scicomp.uni-kl.de