

Using Futhark for a fast, parallel implementation of forward and back projection in algebraic reconstruction methods - A pre-study

Mette Bjerg (zgb585) and Lærke Pedersen (crj405)

November 2, 2018

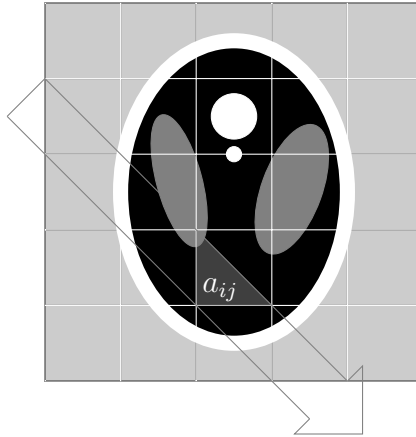


Figure 1: Example of a weighting a_{ij} .

1 Introduction

Computed tomography is the inverse problem of reconstructing an image or volume from its x-ray projections. The x-ray source spins around the object to be analyzed and sends x-rays that hit a detector on the opposite side. The detector shows how much of each x-ray was attenuated when passing through the volume. In this report we will focus on the bottlenecks of an algebraic reconstruction algorithm called the Simultaneous Iterative Reconstruction Algorithm (SIRT), namely forward projection and back projection and investigate how we may use the high level data-parallel language Futhark for implementing a fast version of this algorithm. The reason for choosing this algorithm is that it provides good reconstruction quality under non-optimal circumstances, but unfortunately has very poor performance compared to techniques such as filtered backprojection.

In algebraic reconstruction we solve the problem as a linear system of equations. The main idea is that the process can be modelled as a linear transformation by discretizing the object to be reconstructed into N pixels. First we place a coordinate system with origo at the center of the object to be reconstructed and denote by θ the vector of angles between the positions of the source and origo. For each angle several x-rays are cast from the source. We denote by the vector ρ the signed distances from each line to origo. The data produced by the process is called the sinogram. Then the sinogram values p_i for each (θ_k, ρ_l) are a weighted sum of the attenuations at each pixel f_j that the (θ_k, ρ_l) ray passes through:

$$\sum_{j=1}^N a_{ij} f_j = p_i \quad (1)$$

Where a_{ij} are the weights, corresponding to the fraction of the pixel j that the ray i covers.

Writing all the projections as a column vector \mathbf{p} and the attenuation values to be reconstructed as a column vector \mathbf{f} the weightings are represented as an $M \times N$ matrix \mathbf{A} , we obtain a linear system of the form:

$$\mathbf{p} = \mathbf{A}\mathbf{f} \quad (2)$$

These systems of equations may easily be solved under the right circumstances, where $M = N$. However this is rarely the case. In most real cases $M > N$, i.e. the number of projections is larger than the number of pixels to be reconstructed and the size of the matrix is very large - more about this in the next section.

However, the algebraic reconstruction methods also have some advantages. Since the model closely relates to the real world scenario the weightings can be refactored to take irregularities in the setup, such as differences in beam energies or irregular geometries and missing data into account. Furthermore these methods generally give better image quality than analytic methods when the data is sparse.

A linear system like this is typically solved by minimizing some norm:

$$\|\mathbf{A}\mathbf{f} - \mathbf{p}\| \quad (3)$$

An example is the SIRT algorithm. The action of the matrix \mathbf{A} is called the *forward projection*, and the matrix itself is called the *system matrix, projection matrix* or *radon matrix*. Each row of \mathbf{A} represents the coefficients of the equation for one ray. The action of the transposed projection matrix \mathbf{A}^T on the sinogram vector is called the *backprojection*, and can be vizualized as smearing the projection values across the reconstruction in the direction of the ray, or equivalently summing up the contrubitions of each ray for a given pixel. The idea behind the SIRT algorithm is to forward project the current reconstruction, then subtract this from the original projection data and do a weighted backprojection resulting in a correction factor which can be added to the current reconstruction. The update equeation is:

$$\mathbf{f}^n = \mathbf{f}^{(n-1)} + \mathbf{C}\mathbf{A}^T\mathbf{R}(\mathbf{p} - \mathbf{A}\mathbf{f}^{(n-1)}), \quad (4)$$

where \mathbf{C} and \mathbf{R} are the diagonal matrices containing the inverse column and row sums of the system matrix respectively.

It can be shown that this iterative scheme solves the problem:

$$\mathbf{f}^* = \operatorname{argmin}_{\mathbf{f}} \|\mathbf{p} - \mathbf{A}\mathbf{f}\|_{\mathbf{R}}, \quad (5)$$

where $\|\mathbf{x}\|_{\mathbf{R}} = \mathbf{x}^T \mathbf{R} \mathbf{x}$.

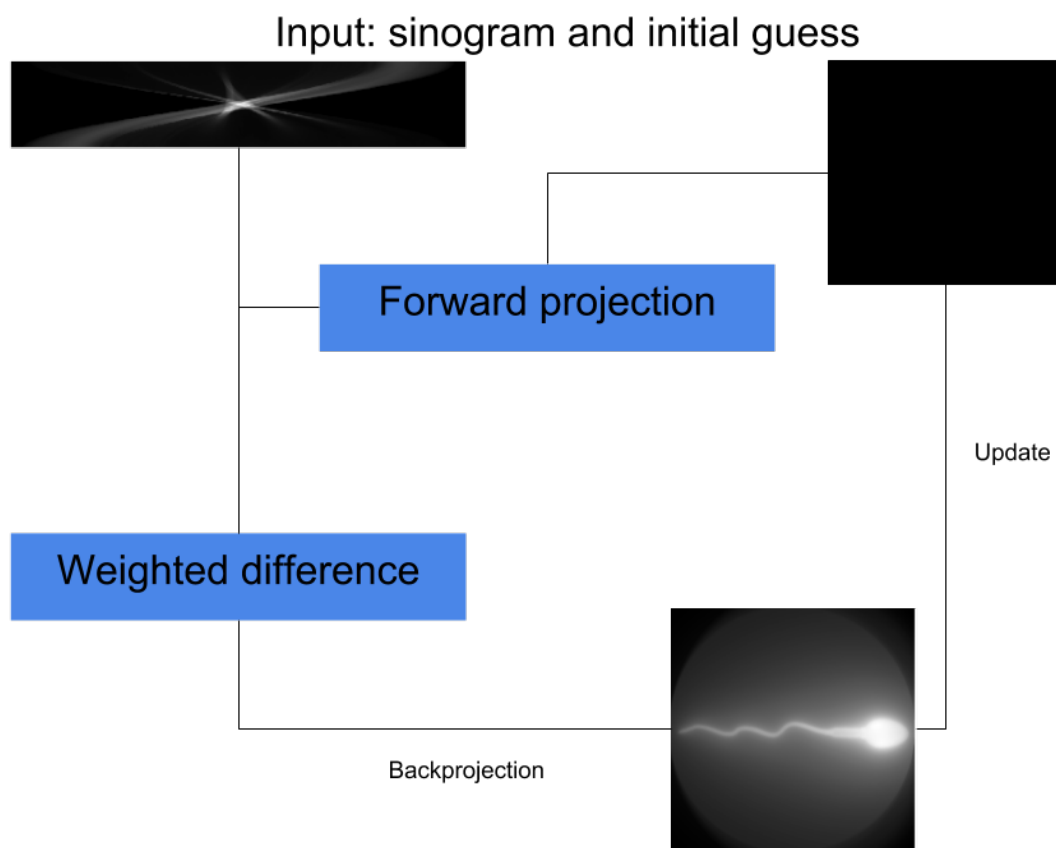


Figure 2: An illustration of the SIRT algorithm.

The backprojection and forward projection operations are standard operations in many iterative algebraic reconstruction methods and are the bottle necks of the algorithms [2]. Therefore, our main focus has been on optimizing these operations.

Different beam geometries exist, such as parallel beams, fan beams and cone beams. The geometry must be considered when constructing the system matrix. However as this study is intended for applications with synchrotron data only parallel beam geometries will be considered. Furthermore, we will only consider reconstructions of 2D images, since this is simply a pre-study and it makes the problem conceptually easier to follow.

2 Computing the projection matrix

One of the main problems when parallelizing the algorithm is that the amount of data in real applications is huge. Images from synchrotrons are generated with detectors of sizes up to about 4000×4000 , with a typical detector size being about 2048×2048 . To accurately generate 3D reconstructions it has been shown that approximately $\frac{\pi \cdot N}{2}$ viewing angles are needed, where N is the size of the detector in one direction [3].

We benchmarked our algorithms by using sizes N ranging from 128 to 4096 and using $\lceil \frac{\pi \cdot N}{2} \rceil$ angles and N lines for each of these sizes to make the problem sizes realistic. However, in the future it might be worth investigating how the reconstructions look when using fewer angles, since one of the strengths of the SIRT algorithm is that it produces good results with few angles.

To solve the large problems it is not possible to store the whole system matrix on the GPU. Let's take as an example the amount of space used to store the matrix for a 3D problem with a $N = 4000$ detector, with $\lceil \frac{\pi \cdot N}{2} \rceil$ angles, and 4000 rays per angle. The matrix is sparse having at most $2 \cdot N - 1$ entries in each row, so instead of storing the whole matrix with all the zeroes, we will consider a semi-sparse representation with one 2 dimensional array containing the data as floats in arrays on each row of size $2 \cdot N - 1$, and one matching 2 dimensional array containing the column indexes of the datapoints. This is also the representation we have used in our implementations. For the example sketched above the number of rows of each matrix will be $\lceil \frac{\pi \cdot 4000}{2} \rceil \cdot 4000$ and the number of columns is $2 \cdot 4000 - 1$. Assuming ints and floats both take up 4 bytes each then storing the matrix requires $4 \cdot 2 \cdot \lceil \frac{\pi \cdot 4000}{2} \rceil \cdot 4000 \cdot (2 \cdot 4000 - 1) \approx 1TB$. This is of course one of the largest problems, but even for a problem with $N = 512$ we're looking at the matrix taking up around 3 – 4GB. Besides the matrix we also need to store the sinogram data which will take approximately 420MB, and the reconstruction taking up around 270MB. Hence even for relatively small problems we're looking at an estimated 5GB of data. GPU memory is expensive, and is typically in the range 2-8GB for consumer end cards, and 10-48GB for high end cards. Thus, the small problems are not able to fit in a standard consumer card, and large problems won't fit on any cards. When we tested our algorithms, we were not able to compute the system matrix on the university's GPU's for sizes larger than 256×256 . There are several ways of getting around this problem. Some problems exhibit symmetries

in the system matrix so that we only have to store part of the matrix. This is already somewhat accounted for in the above calculations. In the example we only considered the matrix for one slice of a 3D reconstruction since it will be equivalent for other slices when using parallel beam geometry. Thus for cone beam geometries, the memory needed would be approximately N times greater. Other symetries involve angle symmetries, where for example a scan from a 45° or 135° angle would contain the same values for mirrored pixels. However these are obviously not generally applicable since they will depend on which angles are used in the reconstruction.

Another more flexible approach is to calculate the system matrix in chunks which is what we have done. For this we started by using the code from a bachelor project in which they did an implementation of finding the system matrix using futhark. We had two different parallel implementations in futhark and a sequential version in python which we compared. The sequential version in python was too slow (in the order of hours) for realistic problem sizes to be worth considering. When we compared the two bachelor projects one seemed to outperform the other both in terms of memory usage and computation time. Since one of the implementations ran in to problems with memory for some of the smallest problems, we decided to keep working with the most promising of the two. We call this algorithm *projectionmatrix_jh*.

The system matrix may be adequately approximated by considering the x-rays as being thin lines and calculating the length of intersections with each pixel in a grid. For simplicity we have only considered square grids with isotropic pixels.

```

1  for ray = 0; ray < numberofrays; ray++ //parallel
2      for pixel = 0; pixel < pixels; pixel++ //parallel
3          if ray intersects pixel:
4              (p1,p2) = intersectionpoints pixel ray
5              A[ray][pixel] = distance p1 p2

```

Figure 3: A looped version of the projection matrix algorithm.

The computation of the system matrix contains two levels of possible parallelization. The implementation from the bachelor project we used had parallelized over the outer loop - i.e for each ray. To compute the intersection lengths a sequential loop was run following each line. The number of rays is very high (6.586.368 for a grid of size $2048 * 2048$), and larger than the number of pixels, this seems a good choice. By following each line, the inner loop only runs $2 * N - 1$ times. From their report it seems they tried a naive implementation of computing the intersection with each pixel in parallel (akin to the pseudocode in 2) but found it to be too slow to consider.

However, when we had an implementation of the foward projection and analyzed the compiled code, we could clearly see that the most time was spend in the kernel consisting of the sequential loop doing grid line intersections. Therefore we decided to spend some more time on the system matrix computations. Futhark does a lot of optimizations, but can not merge code across a loop in a map so our first approach was to attempt to change

the inner loop to a map. However, since the inner loop is inherently sequential because of cross loop dependencies, we did not completely succeed. We separated out the part of the code which computes the intersection point, and did the distance calculations as a map. We call this algorithm *projectionmatrix_map*. Another issue with the *projectionmatrix_jh* is a lot of branches in the inner loop. We also tried to limit the computations in these branches as much as possible in *projectionmatrix_map*.

Since we did not manage to convert the inner loop to a map in *projectionmatrix_jh*, we also attempted a different approach, explained in [1]. It is based on the observation that each line with slope at most 1 intersects at most two rows of the grid in each column of the grid, and opposite for the lines with slope greater than 1. Hence there is potential for exploiting a second level of parallelism by parallelizing over the columns or rows of the grid and finding the nontrivial intersections. They show a method for doing this in time $O(1)$. Unfortunately there is a small bug somewhere in our implementation which seems to be related to the transition between lines with slope > 1 and those with slope < 1 . We did not manage to find the error, but have decided to include the algorithm in the benchmarking anyway, since we believe it is a small error somewhere with no significant influence on the speed of the algorithm. We call this algorithm *projectionmatrix_doubleparallel*.

We benchmarked the algorithms by generating the input, saving it to files and running with *futhatk-bench*. We used the *opencl* compiler, and did 10 runs. From the plot 2 we can see that the version where we exploit inner parallelism is indeed fastest. For the maximum size we were able to run, i.e 256 we got a speedup of about 1.2 compared to *projectionmatrix_jh*.

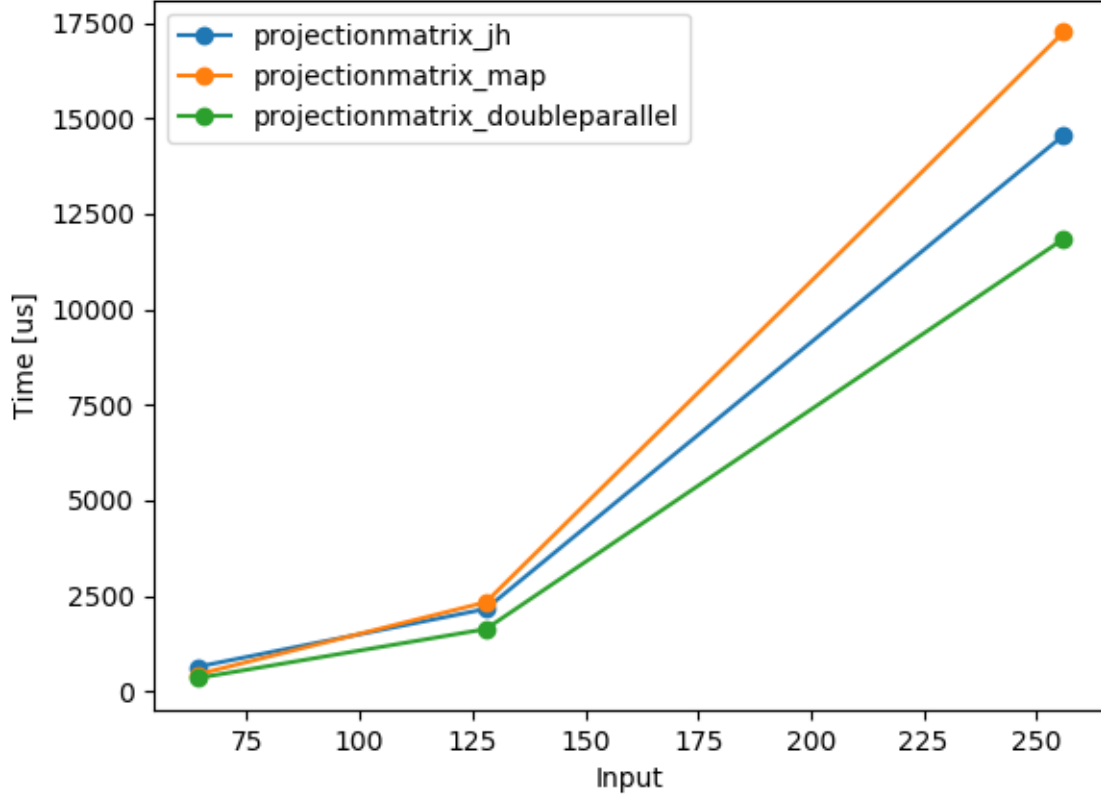


Figure 4: A comparison of the projection matrix algorithms run for gridsizes 64 to 256. Since the universities GPUs only have about 3GB of memory this was the largest size we could run for without chunking up the matrix.

3 Nested parallel forward and back projections

A first approach at forward and backprojection was to do a nested parallel version and using computed chunks of the system matrix.

A looped version of the forward projection with the system matrix cut in *steps* chunks looks like this:


```

1  for step = 0; step < steps; step++
2    A = getRays(raysperstep)
3    for ray = 0; ray < raysperstep; ray++
4      acc = 0.0
5      for p = 0; p<numpixels; p++
6        acc+= A[ray][p]*image[p]
7      FP[step*raysperstep+ray] = acc
8
9  this can be written in futhark like pseudo code as:
10
11 loop (output, step, raysperstep)
12   let A = getRays raysperstep step
13   let partresult = map (\row -> reduce (+) 0 <| map ( \i -> row[i]*
14     vector[i] ) (iota (length row)) ) A
15   in (output++partresult, step, raysperstep)

```

Figure 5: A looped version of the forward projection, where the *raysperstep* should be chosen such that the computations fit in the memory. *step*raysperstep* should equal the total number of rows.

```

1  for step = 0; step < steps; step++
2    A = getRays(raysperstep)
3    AT = A.transpose()
4    for p = 0; p<numpixels; p++
5      acc = 0.0
6      for ray = 0; ray<raysperstep; ray++
7        acc+= AT[p][ray]*sinogram[ray]
8      BP[p] += acc
9
10 this can be written in futhark like pseudo code as:
11
12 loop (output, step, raysperstep)
13   let A = getRays raysperstep step
14   let AT = transpose A
15   let partresult = map (\row -> reduce (+) 0 <| map2 (*) row vect )
16     AT
17   let result = map2 (+) partresult output
18   in (result, step, raysperstep)

```

Figure 6: A looped version of the back projection, where the *raysperstep* should be the largest number possible such that the computations fit in the memory. *step*raysperstep* should equal the total number of rows.

4 Optimizations

Exploiting nested parallelism, is difficult on GPU hardware since the hardware is organized on one, or maybe two, levels that allow threads to communicate via shared scratchpad memory. Hence mapping the application level parallelism to the GPU requires a choice of which level to parallelize, since both level cannot be directly mapped.

One way to get around this problem is to use a flattening transformation. The problem with this is that it will require even more memory usage, and may prevent opportunities for locality optimizations. We tried to do a flattening of our forward and backward projections, since we have multiple levels of parallelism. The idea was to compute the matrix, then transform it to a sparse and flat version and use our sparse matrix vector multiplication from a previous assignment. However, it turned out to be quite complex, and a lot of code was required to transform the matrix to the correct sparse format. Since it later turned out that most of the time spent during the computations was during the matrix computation and we had many issues with running out of memory, this probably wasn't the best approach and we did not pursue it to the end. We report results of semiflat versions. A much more promising approach seems to be feeding the data directly to the matrix computations, and not have to save the matrix data at all but only the end result. We managed to finish an implementation of this approach for forward projection by feeding the data directly to the `projectionmatrix_doubleparallel` version. Unfortunately, it ran out of resources so we need to investigate this further.

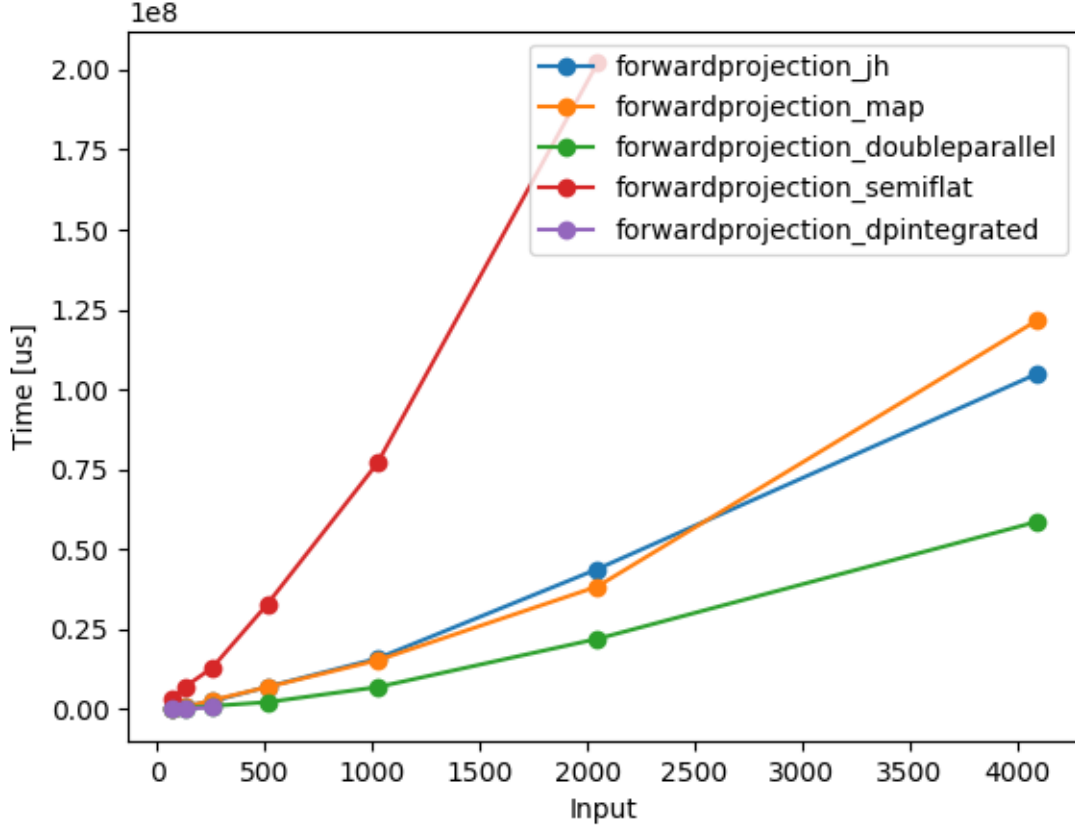


Figure 7: A comparison of the forward projections run with different matrix implementations and the version where the forward projection is integrated in `projectionmatrix_doubleparallel`. The chunk size was 32. We chose this number to fit a CUDA warp. As expected the semiflat version performed rather badly. The implementation using the double parallel version performed best. The rest of the algorithms are nested, using different projection matrix algorithms inside. We had to run our algorithms for only 30 angles as we got memory errors otherwise. This requires further investigation. In our initial tests our stripmined algorithms ran for all sizes, but perhaps we were competing for space with other groups.

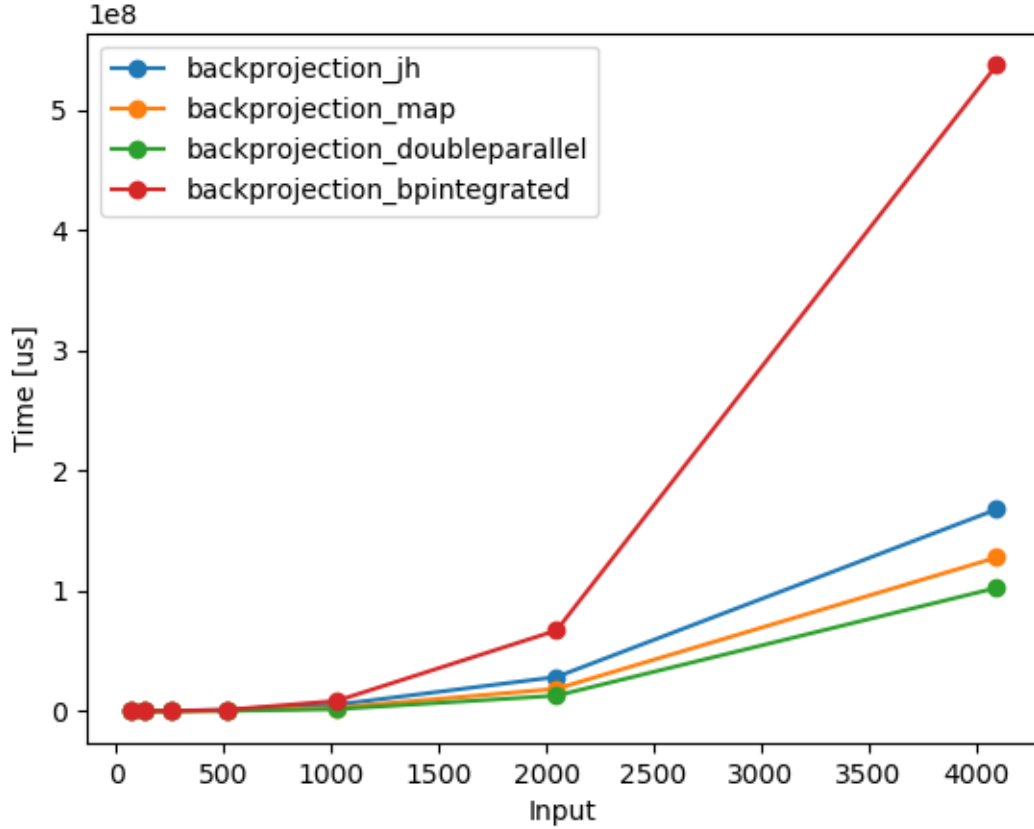


Figure 8: A comparison of the backprojections projections run with different matrix implementations and the version where the forward projection is integrated in `projectionmatrix_doubleparallel`. The chunk size was 32. The implementation using the double parallel version of the matrix performed best again as expected. The integrated version performed very badly, and it would be interesting to investigate why. We had to run our algorithms for only 30 angles as we got memory errors otherwise.

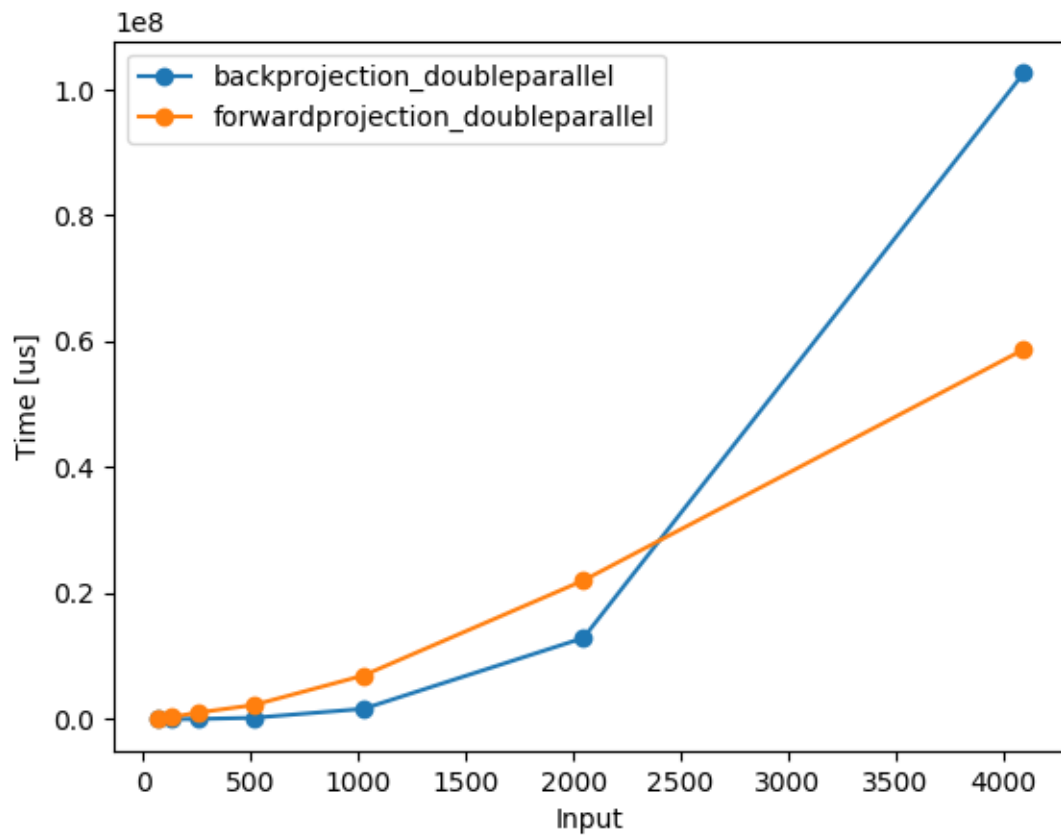
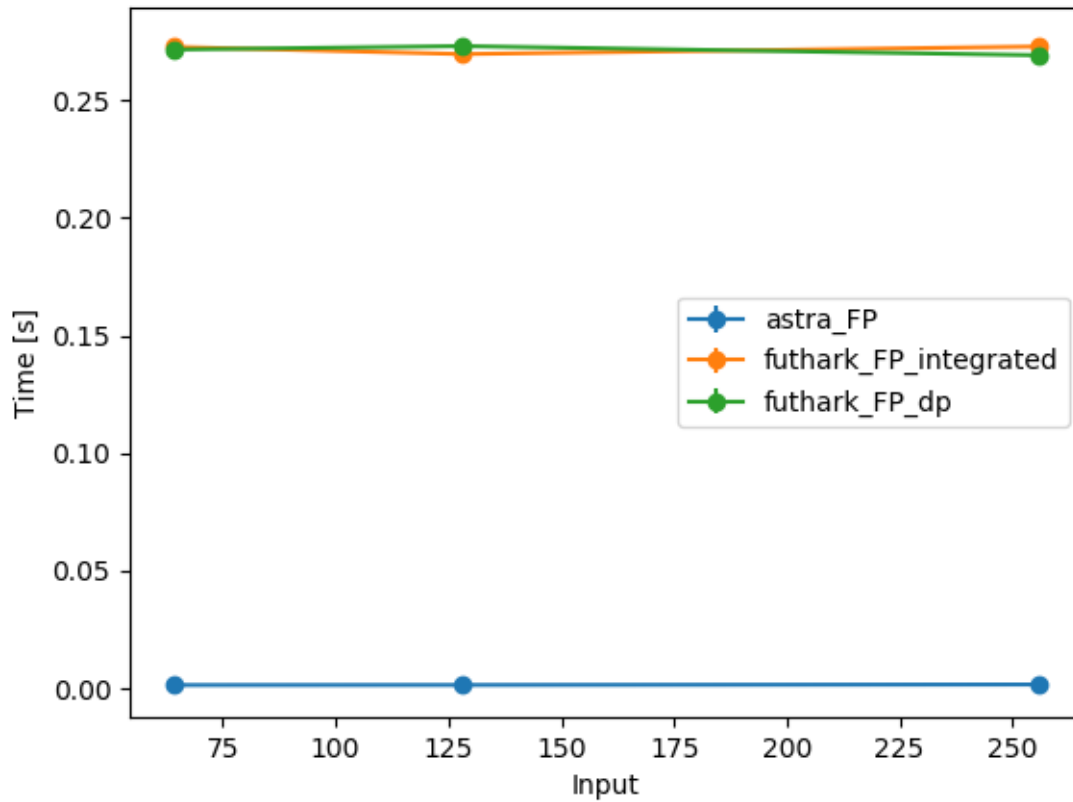


Figure 9: A comparison of backprojection and forward projection. Backprojection seems to be faster for small problem sizes, but scale worse.

5 Comparison to a CUDA implementation

We compared our implementation of forward projection and back projection to implementations from the astra toolbox. However we only compared the algorithms through python, by compiling our algorithms with pyopencl and running them through python and doing the same with the forward and backward rprojection from the atsra toolbox. Ideally the functions should have been compared without the memory overhead.



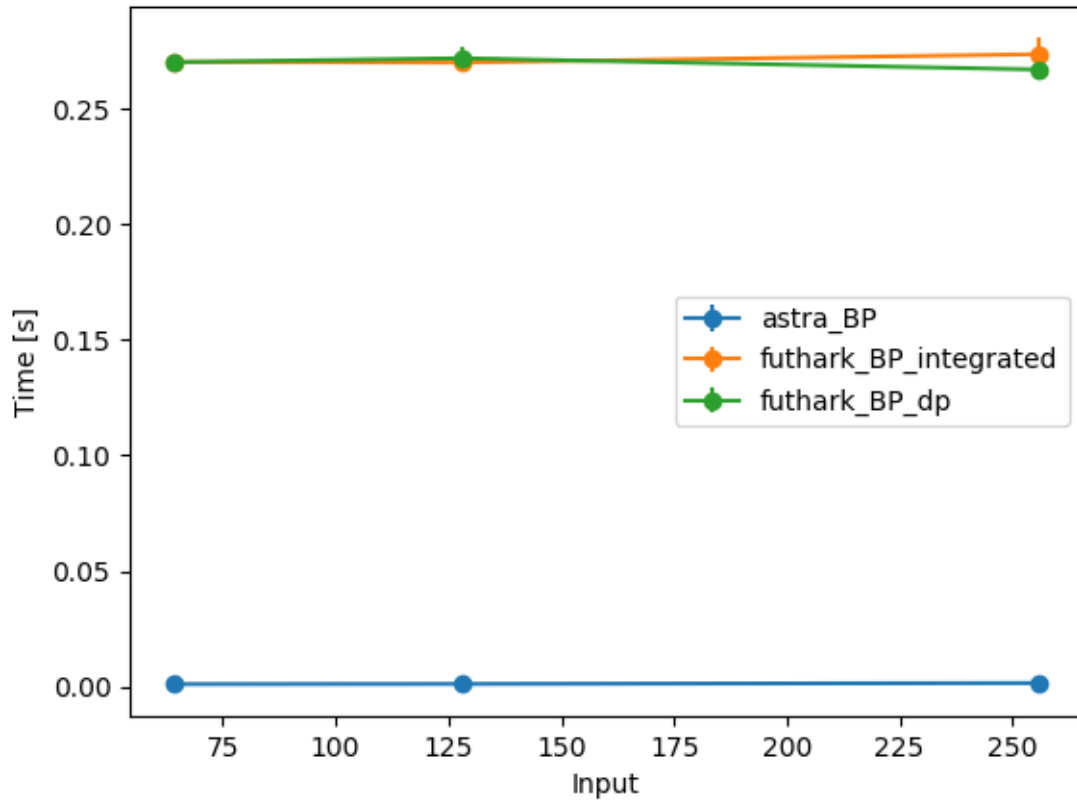


Figure 10: A comparison of the back projections from astra and our fastest futhark algorithms. The futhark algorithms perform much worse, approximately 128 times worse for both forward and backprojection. It would be worth comparing our code with the implemented version from the astra toolbox, to see how they get such a good running time.

6 Code Overview

Our code is primarily comprised of Futhark code, alongside a few python script for data generation, and shell scripts.

6.1 folder structure

- futhark contains the source code files, as well as the files generated during compilation.
- runscripts contains the shell scripts used to run tests and benchmarks
- root folder - the python scripts are located in the root folder, which makes it easier to navigate the folder structure from the python code.
- data and output - data is an, so far, empty folder to store the generated input data, while the output is the same for output data

6.2 Code structure

The actual Futhark code is located in the files `projectionlib.fut`, `matrixlib.fut` and `linelib.fut`. All the `forwardprojection` and `backprojection` files contains main functions to call the libraries for testing. The `projectionlib` contains the actual code for doing the forward-projections and backprojections, as well as a few limited helper functions. The `matrixlib` is focused on generating the system matrices, with code for each way of doing this. Finally the `linelib` contains helper functions for the `matrixlib`.

7 Conclusion

We set out to make a parallel implementation of forward and back projection for algebraic tomographic reconstruction. We found that most of the time spend in our initial nested versions, using code from a bachelor project for generating the system matrix, was spend on calculating the matrix. We therefore concentrated on improving the code for calculating the matrix. We managed to get a slight speedup of about 1.3 times compared to the code from the bachelor project by exploiting a second level of parallelism. When we compare a c compiled version of our futhark code with an opencl compiled version we get a speedup of about 1.3. Strangely enough our c compiled code actually runs slightly faster than the opencl compiled version of the code from the bachelor project. It therefore seems that our new version is better both sequentially and in parallel.

We attempted to compare our implementations with those in a python library called `astra` toolbox with a backend implemented in CUDA. However, we did not time the kernels only, but timed the call from python which has significant overhead. Therefore, it is difficult to compare. The results we got differed on which function was run first, and generally seemed

to vary a lot. To properly time the functions, we would need to go to the astra code and time the kernel only. It seems to perform better than our versions though, and it would be nice to compare the CUDA code with the opencl compiled versions from futhark.

Some of our code may benefit from upcoming block and register tiling in futhark, or streaming possibilities as memory seems to be the biggest limitation. Other possibilities for optimizations could be to consider other datastructures for more sparse storage, or investigate how much precision is needed for good reconstructions. It will also be worth it to look at the branching in the kernels. Flattening does not seem very promising since we already have memory issues and our semiflat versions performed rather badly. Perhaps if the computation of the system matrix was also sparse and flat we would see an improvement.

Overall we were happy to get a slight speedup for the matrix computations, and there is still lots of work that could be done on this project. One thing that needs investigation is why chunking up the matrix does not seem to work as expected as we still ran into memory problems. We look forward to keep working on it with new upcoming features in futhark and to have time to scour the compiled code for further optimizations.

8 Code

8.1 projection lib

```

1  import "futlib/array"
2  import "matrix_lib"
3  open Matrix
4
5  module Projection = {
6
7      --segmented scan with (+) on floats:
8      let sgmSumF32 [n]
9          (flg : [n]i32)
10         (arr : [n]f32) : [n]f32 =
11         let flgs_vals =
12             scan ( \ (f1, x1) (f2,x2) ->
13                 let f = f1 | f2 in
14                 if f2 > 0 then (f, x2)
15                 else (f, x1 + x2) )
16             (0,0.0f32) (zip flg arr)
17         let (_, vals) = unzip flgs_vals
18         in vals
19
20     -- segmented scan with (+) on ints:
21     let sgmSumI32 [n]
22         (flg : [n]i32)
23         (arr : [n]i32) : [n]i32 =
24         let flgs_vals =

```

```

25         scan ( \ (i1, x1) (i2,x2) ->
26             let i = i1 | i2 in
27             if i2 > 0 then (i, x2)
28             else (i, x1 + x2) )
29         (0,0i32) (zip flg arr)
30     let (_, vals) = unzip flgs_vals
31     in vals
32
33 -- unzip for 2D arrays
34 let unzip_d2 (xs: [][f32,i32]): ([][f32],[][i32]) =
35     xs |> map unzip
36         |> unzip
37
38 -- step in radix sort
39 let rsort_step [n] (xs: [n](f32, i32, i32), bitn: i32): [n](f32,
40     i32, i32) =
41     let (data,rays,pixels) = unzip3 xs
42     let unsigned = map(\p -> u32.i32 p) pixels
43     let bits1 = map (\x -> (i32.u32 (x >> u32.i32 bitn)) &
44         1) unsigned
45     let bits0 = map (1-) bits1
46     let idxs0 = map2 (*) bits0 (scan (+) 0 bits0)
47     let idxs1 = scan (+) 0 bits1
48     let offs = reduce (+) 0 bits0
49     let idxs1 = map2 (*) bits1 (map (+offs) idxs1)
50     let idxs = map2 (+) idxs0 idxs1
51     let idxs = map (\x->x-1) idxs
52     in scatter (copy xs) idxs xs
53
54 -- Radix sort algorithm, ascending
55 let rsort [n] (xs: [n](f32, i32, i32)): [n](f32,i32,i32) =
56     loop (xs) for i < 32 do rsort_step(xs,i)
57
58 -- sparse matrix vector multiplication
59 let spMatVctMult [num_elms] [vct_len] [num_rows]
60     (mat_val : [num_elms](i32,f32))
61     (shp_scn : [num_rows]i32)
62     (vct : [vct_len]f32) : [num_rows]f32 =
63     let len = shp_scn[num_rows-1]
64     let shp_inds =
65         map (\i -> if i==0 then 0
66             else unsafe shp_scn[i-1]
67             ) (iota num_rows)
68     let flags = scatter ( replicate len 0)
69         shp_inds ( replicate num_rows 1)
70     let prods = map (\(i,x) -> x*(unsafe vct[i])) mat_val
71     let sums = sgmSumF32 flags prods
72     let mat_inds = map (\i -> i-1) shp_scn
73     in map (\i -> unsafe sums[i]) mat_inds

```

```

74         (mat_vals : [num_rows][num_cols](f32,i32))
75         (vect : []f32) : [num_rows]f32 =
76     map (\row -> reduce (+) 0 <| map (\(v, ind) -> unsafe (if ind
77         == -1 then 0.0 else v*vect[ind])) ) row ) mat_vals
78
79 -- not sparse, nested, matrix vector multiplication for
80 -- backprojection, uses a padded version of the matrix
81 let notSparseMatMult_back [num_rows] [num_cols]
82     (mat_vals : [num_rows][num_cols]f32)
83     (vect : []f32) : [num_rows]f32 =
84     map (\row -> reduce (+) 0 <| map2 (*) row vect ) mat_vals
85
86 -- gets the shape of a matrix - i.e number of entries pr. row
87 -- when mat is in the format [[(d,i),(d,i)...]] where d is data
88 -- and i is column index and -1 means no data
89 let getshp (matrix: [][(f32,i32)]) : []i32 =
90     let dp = unzip_d2 matrix
91     let flagsforindexes = map(\r -> map(\p -> if p == -1
92         then 0 else 1)r)dp.2
93     in map(\f -> reduce (+) 0 f)flagsforindexes
94
95 -- helper function to determine if we entered new segment of
96 -- consecutive numbers in an array.
97 let isnewsegment (i: i32) (arr: []i32) : bool =
98     i!=0 && (unsafe arr[i]) != (unsafe arr[i-1])
99
100 let backprojection_semiplat (rays : []f32)
101     (angles : []f32)
102     (projections : []f32)
103     (gridsize: i32) : []f32=
104     let halfsize = r32(gridsize)/2
105     let entrypoints = convert2entry angles rays halfsize
106     let intersections = map (\(p,sc) -> (lengths gridsize sc
107         .1 sc.2 p)) entrypoints
108     --convert to triples (data,ray,pixel)
109     let triples_tmp = flatten(map(\i -> map(\v -> (v.1, i, v
110         .2))(unsafe intersections[i])) (iota (length
111         intersections)))
112     -- remove none values
113     let triples = filter (\x -> x.3 != -1) triples_tmp
114     -- sort by pixel indexes
115     let pixelsorted = rsort triples
116     -- split int three arrays in order to use pixels only
117     -- for shp
118     let (data,rays,pixels) = unzip3 pixelsorted
119     let num_pixels = length pixels
120     -- contains sum of values where a row ends since columns
121     -- will be rows.
122     let shp_scn_tmp = map (\i -> if (i == num_pixels || (
123         isnewsegment i pixels)) then i else 0) (iota (
124         num_pixels+1))

```

```

113         let shp_scn = filter (\p -> p != 0) shp_scn_tmp
114         let values = map(\x-> (x.2,x.1))pixelsorted
115         in spMatVctMult values shp_scn projections
116
117     -- pads and transposes the matrix, nested, will perform better
118     -- when tiling is improved in futhark
119     let trans_map [m] [n]
120         (matrix : [m][n](f32, i32)) (gridsize: i32): [][]
121         f32 =
122         let rs = gridsize*gridsize
123         let padded = map (\row -> let (vals, inds) = (unzip row)
124             in scatter (replicate rs 0.0f32) inds vals ) matrix
125         in transpose padded
126
127     -- backprojection nested map version.
128     let backprojection_map (angles : []f32)
129         (rays : []f32)
130         (projections : []f32)
131         (gridsize: i32)
132         (stepSize : i32) : []f32=
133     let halfsize = r32(gridsize)/2
134     let entypoints = convert2entry angles rays halfsize
135     let totalLen = (length entypoints)
136     let runLen = (totalLen/stepSize)
137     -- result array
138     let backmat = replicate (gridsize*gridsize) 0.0f32
139     -- stripmined, sequential outer loop, mapped inner
140     let (backmat, _, _, _, _, _, _) =
141         loop (output, run, runLen, stepSize, gridsize,
142             entypoints, totalLen) = (backmat, 0, runLen,
143             stepSize, gridsize, entypoints, totalLen)
144     while ( run < runLen ) do
145         -- if the number of entypoints doesn't line
146         -- perfectly up with the stepsize
147         let step = if (run+1)*stepSize >= totalLen then
148             totalLen - run*stepSize else stepSize
149         -- calc part of matrix, stepSize rows
150         let partmatrix = map (\s -> unsafe (lengths_map
151             gridsize (entypoints[run*stepSize + s].2).1
152             (entypoints[run*stepSize + s].2).2
153             entypoints[run*stepSize + s].1 )) (iota step
154             )
155         -- transpose
156         let transmat = trans_map partmatrix gridsize
157         -- mult
158         let partresult = (notSparseMatMult_back transmat
159             projections[(run*stepSize) : (run*stepSize +
160             step)])
161         -- add
162         let result = (map2 (+) partresult output)

```

```

150             in (result, run+1, runLen, stepSize, gridSize,
151                 entypoints, totalLen)
152
153         in backmat
154
155     let backprojection_jh (angles : []f32)
156         (rays : []f32)
157         (projections : []f32)
158         (gridsize: i32)
159         (stepSize : i32) : []f32=
160     let halfsize = r32(gridsize)/2
161     let entypoints = convert2entry angles rays halfsize
162     let totalLen = (length entypoints)
163     let runLen = (totalLen/stepSize)
164     -- result array
165     let backmat = replicate (gridsize*gridsize) 0.0f32
166     -- stripmined, sequential outer loop, mapped inner
167     let (backmat, _, _, _, _, _, _) =
168         loop (output, run, runLen, stepSize, gridSize,
169             entypoints, totalLen) = (backmat, 0, runLen,
170             stepSize, gridSize, entypoints, totalLen)
171         while ( run < runLen ) do
172             -- if the number of entypoints doesn't line
173             -- perfectly up with the stepsize
174             let step = if (run+1)*stepSize >= totalLen then
175                 totalLen - run*stepSize else stepSize
176             -- calc part of matrix, stepSize rows
177             let partmatrix = map (\s -> unsafe (lengths
178                 gridSize (entypoints[run*stepSize + s].2).1
179                 (entypoints[run*stepSize + s].2).2
180                 entypoints[run*stepSize + s].1 )) (iota step
181                 )
182             -- transpose
183             let transmat = trans_map partmatrix gridSize
184             -- mult
185             let partresult = (notSparseMatMult_back transmat
186                 projections[(run*stepSize) : (run*stepSize +
187                 step)])
188             -- add
189             let result = (map2 (+) partresult output)
190             in (result, run+1, runLen, stepSize, gridSize,
191                 entypoints, totalLen)
192
193         in backmat
194
195     let backprojection_doubleparallel (angles : []f32)
196         (rays : []f32)
197         (projections : []f32)
198         (gridsize: i32)
199         (stepSize : i32) : []f32=
200     let halfsize = r32(gridsize)/2
201     let entryexitpoints = convert2entryexit angles rays
202         halfsize

```

```

188     let totalLen = (length entryexitpoints)
189     let runLen = (totalLen/stepSize)
190     -- result array
191     let backmat = replicate (gridSize*gridSize) 0.0f32
192     -- stripmined, sequential outer loop, mapped inner
193     let (backmat, _, _, _, _, _) =
194         loop (output, run, runLen, stepSize, gridSize,
195             entryexitpoints, totalLen) = (backmat, 0, runLen,
196                 stepSize, gridSize, entryexitpoints, totalLen)
197         while ( run < runLen ) do
198             -- if the number of entypoints doesn't line
199             -- perfectly up with the stepsize
200             let step = if (run+1)*stepSize >= totalLen then
201                 totalLen - run*stepSize else stepSize
202             -- calc part of matrix, stepSize rows
203             let halfgridSize = gridSize/2
204
205             let partmatrix = map(\(ent,ext) -> (flatten(map
206                 (\i ->
207                     calculate_weight ent ext i gridSize
208                     )((-halfgridSize)...(halfgridSize-1)))) (
209                 entryexitpoints[(run*stepSize) : (run*
210                     stepSize + step)]))
211             -- transpose
212             let transmat = trans_map partmatrix gridSize
213             -- mult
214             let partresult = (notSparseMatMult_back transmat
215                 projections[(run*stepSize) : (run*stepSize +
216                     step)])
217             -- add
218             let result = (map2 (+) partresult output)
219             in (result, run+1, runLen, stepSize, gridSize,
220                 entryexitpoints, totalLen)
221         in backmat
222
223 let forwardprojection_doubleparallel (angles : []f32)
224     (rays : []f32)
225     (voxels : []f32)
226     (stepSize : i32) : []f32 =
227     let gridSize = t32(f32.sqrt(r32((length voxels))))
228     let halfgridSize = gridSize/2
229     let entryexitpoints = convert2entryexit angles rays (
230         r32(halfgridSize))
231     let totalLen = (length entryexitpoints)
232     let runLen = (totalLen/stepSize)
233     let testmat = [0f32]
234     let (testmat, _, _, _, _, _) =
235         loop (output, run, runLen, stepSize, gridSize,
236             entryexitpoints, totalLen) = (testmat, 0, runLen,
237                 stepSize, gridSize, entryexitpoints, totalLen)
238         while ( run < runLen ) do

```

```

226         let step = if (run+1)*stepSize >= totalLen then
227             totalLen - run*stepSize else stepSize
228         let partmatrix = map(\(ent,ext) -> (flatten(map
229             (\i ->
230                 calculate_weight ent ext i gridsize
231                 )(-halfgridsize...halfgridsize-1)))) (
232             entryexitpoints[run*stepSize:run*
233                 stepSize+step])
234         let partresult = notSparseMatMult partmatrix
235             voxels
236         in (output++partresult, run+1, runLen, stepSize,
237             gridsize, entryexitpoints, totalLen)
238     in (tail testmat)
239
240 let forwardprojection_jh [r][a][n] (angles : [a]f32)
241     (rays : [r]f32)
242     (voxels : [n]f32)
243     (stepSize : i32) : []f32 =
244 let gridsize = t32(f32.sqrt(r32((length voxels))))
245 let halfsize = r32(gridsize)/2
246 let entrypoints = convert2entry angles rays halfsize
247 let totalLen = (length entrypoints)
248 -- let runLen = if (totalLen/stepSize) == 0 then 1 else
249     (totalLen/stepSize)
250 let runLen = (totalLen/stepSize)
251 let testmat = [0f32]
252 let (testmat, _, _, _, _, _, _) =
253     loop (output, run, runLen, stepSize, gridsize,
254         entrypoints, totalLen) = (testmat, 0, runLen,
255             stepSize, gridsize, entrypoints, totalLen)
256 while ( run < runLen ) do
257     let step = if (run+1)*stepSize >= totalLen then
258         totalLen - run*stepSize else stepSize
259     let partmatrix = map (\s -> unsafe (lengths
260         gridsize (entrypoints[run*stepSize + s].2).1
261         (entrypoints[run*stepSize + s].2).2
262         entrypoints[run*stepSize + s].1 )) (iota step
263         )
264     let partresult = notSparseMatMult partmatrix
265         voxels
266     in (output++partresult, run+1, runLen, stepSize,
267         gridsize, entrypoints, totalLen)
268 in (tail testmat)
269
270 let forwardprojection_map [r][a][n] (angles : [a]f32)
271     (rays : [r]f32)
272     (voxels : [n]f32)
273     (stepSize : i32) : []f32 =
274 let gridsize = t32(f32.sqrt(r32((length voxels))))
275 let halfsize = r32(gridsize)/2
276 let entrypoints = convert2entry angles rays halfsize

```

```

261     let totalLen = (length entrypoints)
262     let runLen = (totalLen/stepSize)
263     let testmat = [0f32]
264     -- stripmined, sequential outer loop, mapped inner
265     let (testmat, _, _, _, _, _) =
266         loop (output, run, runLen, stepSize, gridsize,
                entrypoints, totalLen) = (testmat, 0, runLen,
                stepSize, gridsize, entrypoints, totalLen)
267         while ( run < runLen ) do
268             -- if the number of entrypoints doesn't line
269             -- perfectly up with the stepsize
270             let step = if (run+1)*stepSize >= totalLen then
271                 totalLen - run*stepSize else stepSize
272             -- calc part of matrix, stepSize rows
273             let partmatrix = map (\s -> unsafe (lengths_map
                gridsize (entrypoints[run*stepSize + s].2).1
                (entrypoints[run*stepSize + s].2).2
                entrypoints[run*stepSize + s].1 )) (iota step
                )
274             let partresult = notSparseMatMult partmatrix
                voxels
275             in (output++partresult, run+1, runLen, stepSize,
                gridsize, entrypoints, totalLen)
276         in (tail testmat)
277
278 let forwardprojection_semiplat [r][a][n] (angles : [a]f32)
279     (rays : [r]f32)
280     (voxels : [n]f32)
281     (stepSize : i32) : []f32 =
282     let gridsize = t32(f32.sqrt(r32((length voxels))))
283     let halfsize = r32(gridsize)/2
284     let entrypoints = convert2entry angles rays halfsize
285     let totalLen = (length entrypoints)
286     -- let runLen = if (totalLen/stepSize) == 0 then 1 else
287     (totalLen/stepSize)
288     let runLen = (totalLen/stepSize)
289     let testmat = [0f32]
290     let (testmat, _, _, _, _, _) =
291         loop (output, run, runLen, stepSize, gridsize,
                entrypoints, totalLen) = (testmat, 0, runLen,
                stepSize, gridsize, entrypoints, totalLen)
292         while ( run < runLen ) do
293             let step = if (run+1)*stepSize >= totalLen then
294                 totalLen - run*stepSize else stepSize
295             let intersections = map (\(p,sc) -> (lengths
                gridsize sc.1 sc.2 p)) entrypoints[(run*
                stepSize) : (run*stepSize + step)]
296             let shp = getshp intersections
297             let shp_scn = scan (+) 0 shp
298             let values_tmp = flatten(map(\r -> map(\(d,p)->(
                p,d))r)intersections)

```



```

295         let values = filter (\x -> x.1 != -1) values_tmp
296         let partresult = spMatVctMult values shp_scn
                voxels
297         in (output++partresult, run+1, runLen, stepSize,
                gridsize, entrypoints, totalLen)
298     in (tail testmat)
299
300 let forwardprojection_integrated [r][a][n] (angles : [a]f32)
301     (rays : [r]f32)
302     (voxels : [n]f32)
303     (stepSize : i32) : []f32 =
304 let gridsize = t32(f32.sqrt(r32((length voxels))))
305 let halfsize = gridsize/2
306 let entryexitpoints = convert2entryexit angles rays (
    r32(halfsize))
307 let totalLen = (length entryexitpoints)
308 -- let runLen = if (totalLen/stepSize) == 0 then 1 else
    (totalLen/stepSize)
309 let runLen = (totalLen/stepSize)
310 let testmat = [0f32]
311 let (testmat, _, _, _, _, _, _) =
312     loop (output, run, runLen, stepSize, gridsize,
        entryexitpoints, totalLen) = (testmat, 0, runLen,
        stepSize, gridsize, entryexitpoints, totalLen)
313 while ( run < runLen ) do
314     let step = if (run+1)*stepSize >= totalLen then
        totalLen - run*stepSize else stepSize
315     let partresult = map(\(ent,ext) -> (reduce (+) 0
        (flatten(map (\i ->
316             calculate_fp_val ent ext i gridsize
                voxels
317             )((-halfsize)...(halfsize-1))))))
        entryexitpoints[run*stepSize:run*
        stepSize+step]
318     in (output++partresult, run+1, runLen, stepSize,
        gridsize, entryexitpoints, totalLen)
319 in (tail testmat)
320
321
322 let backprojection_integrated (angles : []f32)
323     (rays : []f32)
324     (projections : []f32)
325     (gridsize: i32)
326     (stepSize : i32) : []f32=
327 let halfsize = gridsize/2
328 let entryexitpoints = convert2entryexit angles rays (
    r32(halfsize))
329 let totalLen = (length entryexitpoints)
330 let runLen = (totalLen/stepSize)
331 -- result array
332 let backmat = replicate (gridsize*gridsize) 0.0f32

```

```

333     -- stripmined, sequential outer loop, mapped inner
334     let (backmat, _, _, _, _, _) =
335         loop (output, run, runLen, stepSize, gridsize,
336             entryexitpoints, totalLen) = (backmat, 0, runLen,
337             stepSize, gridsize, entryexitpoints, totalLen)
338         while ( run < runLen ) do
339             -- if the number of entypoints doesn't line
340             -- perfectly up with the stepsize
341             let step = if (run+1)*stepSize >= totalLen then
342                 totalLen - run*stepSize else stepSize
343             let partmatresult = map(\j ->
344                 (flatten(map (\i ->
345                     calculate_bp_val (unsafe
346                         entryexitpoints[run*stepSize+j]).1 (
347                             unsafe entryexitpoints[run*stepSize+
348                                 j]).2 i gridsize (unsafe projections
349                                 [j])
350                     )((-halfsize)...(halfsize-1)))))) (iota step)
351             let transp = trans_map partmatresult gridsize
352             let partresult = map (\row -> reduce (+) 0 row)
353                 transp
354             -- add
355             let result = (map2 (+) partresult output)
356             in (result, run+1, runLen, stepSize, gridsize,
357                 entryexitpoints, totalLen)
358         in backmat
359 }

```

8.2 matrix lib

```

1  import "line_lib"
2  open Lines
3  module Matrix =
4  {
5      let calculate_fp_val(ent: point)
6          (ext: point)
7          (i: i32)
8          (N: i32)
9          (pixels: []f32) : []f32 =
10         let Nhalf = N/2
11         -- handle all lines as slope < 1 reverse the others
12         let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
13         let reverse = f32.abs(slope) > 1
14         let gridentry = if reverse then (if slope < 0 then (-ent.2,
15             ent.1) else (-ext.2,ext.1)) else ent
16         let k = if reverse then (-1/slope) else slope
17         --- calculate stuff
18         let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
19             Nhalf)
19         let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +

```

```

20         r32(Nhalf)
21     let Ypixmap = t32(f32.floor(ymin))
22     let Ypixplus = t32(f32.floor(yplus))
23     let baselength = f32.sqrt(1+k*k)
24     -- in [(baselength,Ypixmap),(baselength,Ypixplus)]
25     let Ypixmax = i32.max Ypixmap Ypixplus
26     let ydiff = yplus - ymin
27     let yminfact = (r32(Ypixmax) - ymin)/ydiff
28     let yplusfact = (yplus - r32(Ypixmax))/ydiff
29     let iindex = i+Nhalf
30     -- index calculated wrong for reversed lines i think
31     let pixmin = if reverse then (N-iindex-1)*N+Ypixmap else
32         iindex+Ypixmap*N
33     let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
34         iindex+Ypixplus*N
35     let lymin = yminfact*baselength
36     let lyplus = yplusfact*baselength
37     let min = if (pixmin >= 0 && pixmin < N ** 2) then
38         (if Ypixmap == Ypixplus then (unsafe baselength*pixels
39             [pixmin]) else (unsafe lymin*pixels[pixmin]))
40         else 0
41     let plus = if (pixplus >= 0 && pixplus < N ** 2) then
42         (if Ypixmap == Ypixplus then 0 else (unsafe lyplus*
43             pixels[pixmin]))
44         else 0
45     in [min,plus]
46
47 let calculate_bp_val(ent: point)
48     (ext: point)
49     (i: i32)
50     (N: i32)
51     (proj_val: f32) : [(f32,i32) =
52     let Nhalf = N/2
53     -- handle all lines as slope < 1 reverse the others
54     let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
55     let reverse = f32.abs(slope) > 1
56     let gridentry = if reverse then (if slope < 0 then (-ent.2,
57         ent.1) else (-ext.2,ext.1)) else ent
58     let k = if reverse then (-1/slope) else slope
59
60     --- calculate stuff
61     let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
62         Nhalf)
63     let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +
64         r32(Nhalf)
65     let Ypixmap = t32(f32.floor(ymin))
66     let Ypixplus = t32(f32.floor(yplus))
67     let baselength = f32.sqrt(1+k*k)
68     -- in [(baselength,Ypixmap),(baselength,Ypixplus)]
69     let Ypixmax = i32.max Ypixmap Ypixplus
70     let ydiff = yplus - ymin

```

```

63     let yminfact = (r32(Ypixmax) - ymin)/ydiff
64     let yplusfact = (yplus - r32(Ypixmax))/ydiff
65     let iindex = i+Nhalf
66     -- index calculated wrong for reversed lines i think
67     let pixmin = if reverse then (N-iindex-1)*N+Ypixmin else
        iindex+Ypixmin*N
68     let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
        iindex+Ypixplus*N
69     let lymin = yminfact*baselength
70     let lyplus = yplusfact*baselength
71     let min = if (pixmin >= 0 && pixmin < N ** 2) then
72         (if Ypixmin == Ypixplus then ((baselength*proj_val),
        pixmin) else (lymin*proj_val,pixmin))
73         else (-1f32,-1i32)
74     let plus = if (pixplus >= 0 && pixplus < N ** 2) then
75         (if Ypixmin == Ypixplus then (-1f32,-1i32) else ((
        lyplus*proj_val), pixmin))
76         else (-1f32,-1i32)
77     in [min,plus]
78
79
80 --- DOUBLE PARALLEL
81 -- function which computes the weight of pixels in grid_column
    for ray with entry/exit p
82 let calculate_weight(ent: point)
83     (ext: point)
84     (i: i32)
85     (N: i32) : [](f32,i32) =
86     let Nhalf = N/2
87     -- handle all lines as slope < 1 reverse the others
88     let slope = (ext.2 - ent.2)/(ext.1 - ent.1)
89     let reverse = f32.abs(slope) > 1
90     let gridentry = if reverse then (if slope < 0 then (-ent.2,
        ent.1) else (-ext.2,ext.1)) else ent
91     let k = if reverse then (-1/slope) else slope
92
93     --- calculate stuff
94     let ymin = k*(r32(i) - gridentry.1) + gridentry.2 + r32(
        Nhalf)
95     let yplus = k*(r32(i) + 1 - gridentry.1) + gridentry.2 +
        r32(Nhalf)
96     let Ypixmin = t32(f32.floor(ymin))
97     let Ypixplus = t32(f32.floor(yplus))
98     let baselength = f32.sqrt(1+k*k)
99     -- in [(baselength,Ypixmin),(baselength,Ypixplus)]
100    let Ypixmax = i32.max Ypixmin Ypixplus
101    let ydiff = yplus - ymin
102    let yminfact = (r32(Ypixmax) - ymin)/ydiff
103    let yplusfact = (yplus - r32(Ypixmax))/ydiff
104    let lymin = yminfact*baselength
105    let lyplus = yplusfact*baselength

```

```

106     let iindex = i+Nhalf
107     -- index calculated wrong for reversed lines i think
108     let pixmin = if reverse then (N-iindex-1)*N+Ypixmin else
109                 iindex+Ypixmin*N
110     let pixplus = if reverse then (N-iindex-1)*N+Ypixplus else
111                 iindex+Ypixplus*N
112     let min = if (pixmin >= 0 && pixmin < N ** 2) then
113                 (if Ypixmin == Ypixplus then (baselength,pixmin) else
114                  (lymin,pixmin))
115                 else (-1f32,-1i32)
116     let plus = if (pixplus >= 0 && pixplus < N ** 2) then
117                 (if Ypixmin == Ypixplus then (-1f32,-1i32) else (
118                  lyplus,pixplus))
119                 else (-1f32,-1i32)
120     in [min,plus]
121
122 -- assuming gridsize even
123 let weights_doublepar(angles: []f32) (rays: []f32) (gridsize:
124 i32): [][](f32,i32) =
125     let halfgridsize = gridsize/2
126     let entryexitpoints = convert2entryexit angles rays (r32(
127     halfgridsize))
128     in map(\(ent,ext) -> (flatten(map (\i ->
129         calculate_weight ent ext i gridsize
130         )((-halfgridsize)...(halfgridsize-1))))
131         entryexitpoints
132
133 --- JH VERSION
134 let lengths      (grid_size: i32)
135                  (sint: f32)
136                  (cost: f32)
137                  (entry_point: point): [](f32, i32) =
138
139     let horizontal = cost == 0
140     let vertical = f32.abs(cost) == 1
141     let slope = cost/(-sint) -- tan(x+90) = -cot(x) = slope
142     since the angles are the normals of the line
143
144     let size = r32(grid_size)
145     let halfsize = size/2.0f32
146
147     let A = replicate (t32(size*2f32-1f32)) (-1f32, -1)
148
149     let y_step_dir = if slope < 0f32 then -1f32 else 1f32
150     let anchorX = f32.floor(entry_point.1) + 1f32
151     let anchorY = if y_step_dir == -1f32
152                     then f32.ceil(entry_point.2) - 1f32
153                     else f32.floor(entry_point.2) + 1f32
154
155     let (A, _, _, _, _) =
156         loop (A, focusPoint, anchorX, anchorY, write_index) = (A,

```

```

149     entry_point, anchorX, anchorY, 0)
150 while ( isInGrid halfsize y_step_dir focusPoint ) do
151     --compute index of pixel in array by computing x
152     --component and y component if
153     --center was at bottom left corner (add halfsize), and
154     --add them multiplying y_comp by size
155     let y_floor = f32.floor(halfsize+focusPoint.2)
156     let y_comp =
157         if (y_step_dir == -1f32 && focusPoint.2 - f32.floor(
158             focusPoint.2) == 0f32)
159             then y_floor - 1f32
160             else y_floor
161     let x_comp= f32.floor(halfsize+focusPoint.1)
162     let index = t32(x_comp+size*y_comp)
163
164     --compute the distances using the difference travelled
165     --along an axis to the
166     --next whole number and the slope or inverse slope
167     let dy = if vertical then 1f32 else if horizontal then 0
168             f32 else (anchorX-focusPoint.1)*slope
169     let dx = if vertical then 0f32 else if horizontal then 1
170             f32 else (anchorY-focusPoint.2)*(1/slope)
171     let p_anchor_x = (anchorX, focusPoint.2+dy)
172     let p_anchor_y = (focusPoint.1+dx, anchorY)
173
174     let dist_p_x = distance focusPoint p_anchor_x
175     let dist_p_y = distance focusPoint p_anchor_y
176
177 in
178     if horizontal then
179         unsafe let A[write_index] = (dist_p_x, index)
180         in (A, p_anchor_x, anchorX + 1f32, anchorY,
181             write_index+1)
182     else if vertical then
183         unsafe let A[write_index] = (dist_p_y, index)
184         in (A, p_anchor_y, anchorX, anchorY + y_step_dir,
185             write_index+1)
186     else
187         if (f32.abs(dist_p_x - dist_p_y) > 0.000000001f32)
188         then
189             if ( dist_p_x < dist_p_y )
190             then
191                 unsafe let A[write_index] = (dist_p_x, index)
192                 in (A, p_anchor_x, anchorX + 1f32, anchorY,
193                     write_index+1)
194             else
195                 unsafe let A[write_index] = (dist_p_y, index)
196                 in (A, p_anchor_y, anchorX, anchorY + y_step_dir,
197                     write_index+1)
198         else
199             unsafe let A[write_index] = (dist_p_x, index)

```

```

189             in (A, p_anchor_x, anchorX + 1f32, anchorY +
190                 y_step_dir, write_index+1)
191
192         in A
193
194     let weights_jh    (angles: []f32)
195                       (rays: []f32)
196                       (gridsize: i32) : [][](f32,i32) =
197         let halfsize = r32(gridsize)/2
198         let entrypoints = convert2entry angles rays halfsize
199         in map (\(p,sc) -> (lengths gridsize sc.1 sc.2 p))
200             entrypoints
201
202     -----MAP
203
204     let index (focusPoint: point) (halfsize: f32) (y_step_dir): i32
205     =
206         let y_floor = f32.floor(halfsize+focusPoint.2)
207         let y_comp =
208             if (y_step_dir == -1f32 && focusPoint.2 - f32.floor(
209                 focusPoint.2) == 0f32)
210             then y_floor - 1f32
211             else y_floor
212         let x_comp = f32.floor(halfsize+focusPoint.1)
213         in t32(x_comp+(halfsize*2f32)*y_comp)
214
215     -- let nextpointonline (vertical: bool) (horizontal: bool) (
216         anchorX: point) (anchorY: point) (slope: f32) (focusPoint:
217         point): point or array of points and lengths
218
219     let nextpointonline (slope: f32) (vertical: bool) (focusPoint:
220     point): point =
221         let y_step_dir = if slope < 0f32 then -1f32 else 1f32
222         let anchorX = if vertical then focusPoint.1 else f32.floor(
223             focusPoint.1) + 1f32
224         let anchorY = if slope == 0 then focusPoint.2 else if
225             y_step_dir == -1f32
226             then f32.ceil(focusPoint.2) - 1f32
227             else f32.floor(focusPoint.2) + 1f32
228         let dy = if slope == 1 then 1f32 else if slope == 0 then 0
229             f32 else (anchorX-focusPoint.1)*slope
230         let dx = if slope == 1 then 0f32 else if slope == 0 then 1
231             f32 else (anchorY-focusPoint.2)*(1/slope)
232         let p_anchor_x = (anchorX, focusPoint.2+dy)
233         let p_anchor_y = (focusPoint.1+dx, anchorY)
234         in if p_anchor_x.1 < p_anchor_y.1 then p_anchor_x else
235             p_anchor_y
236
237     let getFocusPoints (entryPoint: point) slope vertical halfsize
238         y_step_dir =
239         let A = replicate (t32(2f32*halfsize*2f32-1f32)) (f32.
240             lowest, f32.lowest)
241         let (A, _, _) =

```

```

225         loop (A, focusPoint, write_index) = (A, entryPoint, 0)
226         while ( isInGrid halfsize y_step_dir focusPoint ) do
227             let nextpoint = (nextpointonline slope vertical
                               focusPoint)
228             in unsafe let A[write_index] = focusPoint
229             in (A, nextpoint, write_index+1)
230         in A
231
232     let lengths_map
233         (grid_size: i32)
234         (sint: f32)
235         (cost: f32)
236         (entryPoint: point): [](f32, i32) =
237
238     let vertical = f32.abs(cost) == 1
239     let slope = cost/(-sint)
240
241     let size = r32(grid_size)
242     let halfsize = size/2.0f32
243
244     let y_step_dir = if slope < 0f32 then -1f32 else 1f32
245     let focuspoints = (getFocusPoints entryPoint slope vertical
                        halfsize y_step_dir)
246     --for all focuspoints, save index and distance
247     let mf = map(\i ->
248         let ind = if !(isInGrid halfsize y_step_dir (unsafe
            focuspoints[i])) then -1 else index (unsafe
            focuspoints[i]) halfsize y_step_dir
249         let dist = if isInGrid halfsize y_step_dir (unsafe
            focuspoints[i+1]) then (unsafe (distance
            focuspoints[i] focuspoints[i+1])) else 0.0f32
250         in (dist, ind)
251     ) (iota ((length focuspoints)-1))
252     in mf
253
254     let weights_map (angles: []f32)
255                     (rays: []f32)
256                     (gridsize: i32) : [][](f32,i32) =
257     let halfsize = r32(gridsize)/2
258     let entrypoints = convert2entry angles rays halfsize
259     in map (\(p,sc) -> (lengths_map gridsize sc.1 sc.2 p))
        entrypoints
260 }

```

References

- [1] Hao Gao. Fast parallel algorithms for the x-ray transform and its adjoint. *Medical physics*, 39(23127102):7110–7120, November 2012.
- [2] Y. Long, J. A. Fessler, and J. M. Balter. 3d forward and back-projection for x-ray ct using separable footprints. *IEEE Transactions on Medical Imaging*, 29(11):1839–1850,

Nov 2010.

- [3] F. Natterer. *The Mathematics of Computerized Tomography*. Society for Industrial and Applied Mathematics, 2001.
- [4] Z. Xue, L. Zhang, and J. Pan. A new algorithm for calculating the radiological path in ct image reconstruction. In *Proceedings of 2011 International Conference on Electronic Mechanical Engineering and Information Technology*, volume 9, pages 4527–4530, Aug 2011.